Improving Compute & Data Efficiency of Flexible Architectures

Improving Compute & Data Efficiency of Flexible Architectures

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op donderdag 22 september 2022 om 13:30 uur

door

Luc Johannes Wilhelmus Waeijen

geboren te Roermond

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir. J. Voeten
promotor:	prof.dr. H. Corporaal
copromotor:	dr.ir. Y. He (Reconova Technologies Co. Ltd.)
leden:	dr.ir. S. Stuijk
	prof.dr.ir M. Verhelst (Katholieke Universiteit Leuven)
	prof.dr. R.V. van Nieuwpoort (Universiteit van Amsterdam)
adviseurs:	dr.ir. M. Peemen (Thermo Fisher Scientific - FEI)
	dr.ir. S. Sioutas (GrAI Matter Labs)

Het onderzoek dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Improving Compute & Data Efficiency of Flexible Architectures

Luc Johannes Wilhelmus Waeijen

Committee:

prof.dr. H. Corporaal	(*promotor*, TU Eindhoven)
dr.ir. Y. He	(*copromotor*, Reconova Technologies Co. Ltd.)
prof.dr.ir. J. Voeten	(*chairman*, TU Eindhoven)
dr.ir. S. Stuijk	(TU Eindhoven)
prof.dr.ir M. Verhelst	(Katholieke Universiteit Leuven)
prof.dr. R.V. van Nieuwpoort	(Universiteit van Amsterdam)
dr.ir. M. Peemen	(Thermo Fisher Scientific - FEI)
dr.ir. S. Sioutas	(GrAI Matter Labs)

Improving Compute & Data Efficiency of Flexible Architectures

Luc Johannes Wilhelmus Waeijen

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN 978-90-386-5565-9
NUR-code: 959

Printed by Ridderprint

Abstract

Man's ambition to construct machinery that can match, or even exceed, his own intelligence has driven over half a century of research into computer architectures. Enabled by an exponential increase in compute power, the past decades bore witness to several major advancements towards achieving such *artificial intelligence*. Exemplary are the many applications that have started to emerge in our daily lives: autonomous vehicles navigate the roads on our behalf, digital assistants manage our agendas, and recommender systems automatically determine the next products we should want to buy. However, the silicon-based brains that execute these *deep-learning* applications still leave much to be desired, both in intelligence and *energy efficiency*, when compared to their organic counterparts. This relatively poor energy efficiency is in fact what limits the acceptable compute complexity of deep-learning applications, limiting their potential in practice. To advance the capabilities of deep-learning applications it is therefore imperative to improve the energy efficiency of the machines that evaluate them. This is particularly challenging in the highly volatile context of deep-learning algorithms, which require a certain degree of flexibility from the compute architecture to effectively adapt to modifications in the core algorithms.

This thesis focusses on *improving the energy efficiency of modern computing machinery in general, and for deep-learning applications in particular*. The approach is split into three distinct parts: improving *compute efficiency*, improving *data efficiency*, and finally a study into the *flexibility of compute architectures*. Combined these parts contribute towards increased energy efficiency, i.e., the number of computations per Joule, achievable in the evaluation of deep-learning applications mapped to flexible compute platforms.

The first step towards improved compute efficiency is one of a computer-organisational nature. From the conceptual Turing machine to modern-day specialised processors, a common denominator between all computers is the fundamental cycle of instruction selection and evaluation, of which both steps typically consume energy. A classic approach to reduce the energy required for instruction selection is to evaluate a single instruction for multiple data elements (SIMD), effectively amortizing the cost. This concept is taken to the extreme in a wide-single instruction multiple data (SIMD) machine and, crucially, combined with an explicit datapath to reduce the energy consumption by avoiding

accesses to register files in each lane of the SIMD machine. Experimental results show that explicit datapaths are particularly effective in wide-SIMD machines, improving the compute energy efficiency of the baseline machine by 27 % on average. Experiments show that, on a 40 nm node, the proposed 128-lane SIMD machine improves the compute efficiency by a factor 2× on average compared to a reduced instruction set computer (RISC), while accelerating 3×3 convolution by about 200×.

The second step towards improved compute efficiency focusses on the multiply accumulate (MAC) operation which, being a cornerstone of deep-learning applications, forms a prime candidate for optimisation. The computation efficiency of this operation is improved by two techniques: data-width aware multiplication, and parallel accumulation without dedicated hardware. A data-width aware multiplier is proposed, which dynamically improves energy efficiency when its operands do not occupy the full width of the datapath. For deep-learning networks, where the filter parameters and feature-map data often consist of small numbers, this technique has the potential to improve the energy efficiency of a multiplication up to a factor 2.25× while preserving support for wide parameters.

Application of these data-width aware multipliers has an area penalty, which in part is compensated by a parallel-reduction technique applied in the context of the proposed wide-SIMD machine. Amongst other reduction operations, parallel accumulation is supported by this technique, saving 6.8 % in silicon area compared to a dedicated adder-tree without compromising the energy efficiency. This concludes the compute efficiency optimisations incorporated in this thesis.

Part two of the thesis introduces advanced scheduling techniques to improve the *data efficiency* of deep-learning applications, i.e., minimising the average energy spent per data use. To achieve this, the data-memory hierarchy has to be exploited efficiently, serving as many data uses as possible from small, local memories. The number of data uses is determined by the application, but within the given data dependencies they can be scheduled in various orders. For deep-learning applications with their associative MAC operations, this scheduling space is vast. To assist the selection of those schedules that efficiently use a particular memory system, the scheduling space and its data efficiency are precisely modelled. In contrast to most existing approaches, this scheduling space includes layer fusion, which significantly improves the data efficiency of the schedule for large networks on platforms with large local memories. Experimental results show that compared to a straightforward schedule the energy required for evaluating a network can be reduced by an order of magnitude.

To efficiently support future developments in deep learning a degree of flexibility is required from the compute system. However, flexibility as a measure is ill-defined, which obstructs appropriately balancing the architecture. The third and final part of this thesis focusses on investigating and properly defining the concept of *compute flexibility*. In this thesis an initial attempt is made to qualify and quantify flexibility, and investigate its relation to energy efficiency and other system properties in a rigorous manner. This definition and investigation may deepen the understanding of computer architectures in general, and ultimately lead to new, even more efficient designs.

Combined these three parts improve the compute and data efficiency of programmable compute architectures, and provide an initial guideline on objectively determining the flexibility of architectures.

Contents

Chapter 1

Introduction

"Give me a place to stand and with a lever I will move the world"[1] This quote is attributed to Archimedes [182], who realised that with a long enough lever, and a solid foundation to place it on, he could move the world. But whereas the physical world may be moved by a lever and fulcrum, the modern digital world does not yield to such instruments. Instead, it can be argued that to move the modern world, a fast and powerful computer is required.

The introduction of ENIAC, the first programmable, general-purpose computer in 1945 has set off a digital revolution that is still ongoing today. Figure 1.1 shows, besides how overpriced the iPad2 was for its processing power, how the inflation corrected price for compute has decreased exponentially since 1945. To put this in perspective, if the typical worker in 1982 wanted to purchase something with the compute power of an iPad2, it would have cost more than 360 years worth of wages [117]. This rapid price decrease has propelled computers out of the lab and into the core of our daily lives and society. Objects around us are increasingly equipped with flexible processors that handle a wide range of tasks. Watches tell us how we are feeling, and if we are getting enough quality sleep. Noise cancelling headphones ensure we do not have to talk to strangers in public transport. Voice-controlled assistants save us the effort of pressing a button if we want to turn on the lights.

Increased compute capacity was also a catalyst for the neural network revolution that started in 2012 [81]. Since the conceptualisation of computing machinery, man has aspired to build intelligent machines [151]. Naturally this desire has triggered research into mimicking human brains, which are the most intelligent of brains[2]. Building upon the first mathematical model of an artificial neuron by McCulloch and Pitts [101] many computational models for artificial neural networks (ANNs) have been proposed over the past century as shown in figure 1.2. During this period the popularity of ANNs has heavily fluctuated, as progress has

[1] "Πᾷ βῶ, καὶ χαριστίωνι τὰν γᾶν κινήσω πᾶσαν"

[2] according to the human brain

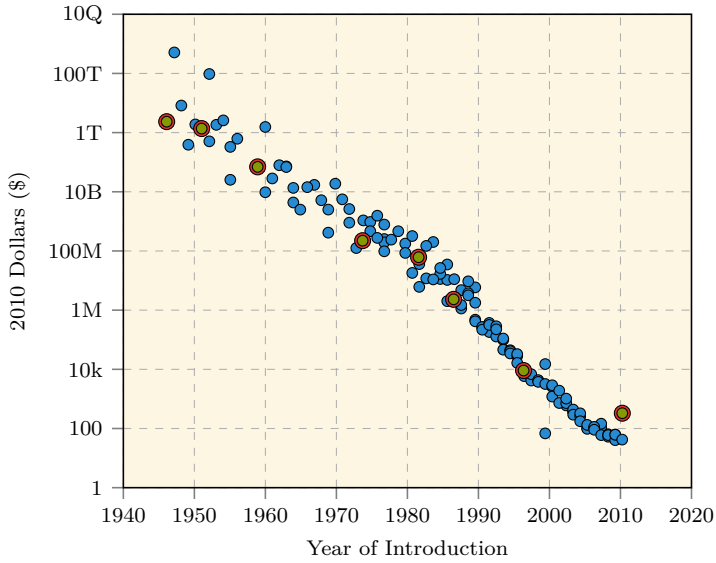


Figure 1.1: Cost of computing power equal to an iPad2 (1600 MIPS) normalised to 2010 dollars based on release price (data source: The Hamilton Project [117]).

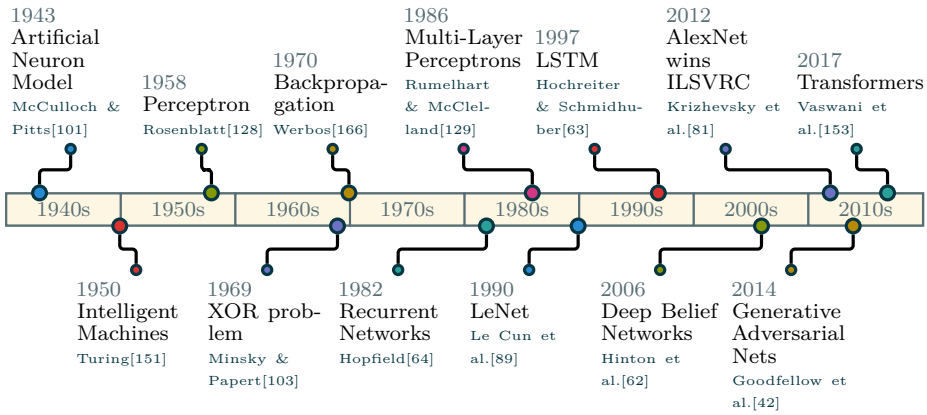


Figure 1.2: Timeline of selected important events in artificial neural network research.

halted on several occasions when barriers needed to be overcome. An important barrier was the required compute capacity for neural network models of sufficient complexity to perform useful tasks. The latest surge of interest in artificial neural networks has been sparked by the work of Krizhevsky et al., who in 2012 greatly outperformed their direct competition in the ImageNet large scale visual recognition challenge (ILSVRC) [85] using an ANN coined "AlexNet" [81]. In this challenge teams competed to automatically classify images into one of a thousand categories. AlexNet improved classification accuracy by 70 % relative to the best competitor using classical methods. Part of their success is directly attributed to the availability of flexible commercial of the shelf (COTS) graphics processing units (GPUs) that could be programmed to perform the training of AlexNet [81]. Since then intensified research into ANNs has yielded a plethora of applications: medical diagnosis, speech recognition, recommender systems, and image enhancement in televisions and games to name but a few.

Despite such successes, many applications today are still restricted by a lack of compute power. Especially the compute complexity of modern ANNs is rather excessive compared to traditional applications, which incurs high costs in terms of processing time and energy requirements. This limitation holds in particular for embedded devices which have strict power and energy budgets. Some application designers attempt to work around these restrictions by offloading work to centralized compute capacity in data centres. This solution is far from ideal however, as transferring workload inevitably also requires energy

for communication, and the added delay between input and result may be unacceptable for latency-sensitive applications. Also, from a privacy perspective it is undesirable for certain applications to send everything across the world-wide web to a server 'somewhere'. This approach is merely shifting rather than solving the problem, since centralized compute inherently does not scale. Furthermore, the compute load on central machines is also pushed to the limit as demonstrated by the expensive cooling installed in the typical data centre.

To enable more complex algorithms on either embedded devices or in centralized data centres, it is imperative to improve the energy efficiency of computing machinery without (excessively) sacrificing speed or flexibility. Only by performing more useful work per Joule can complex algorithms be kept within their energy budgets. The objective of the work presented in this thesis is to improve the energy efficiency of flexible computing systems in general, with a focus on artificial neural network evaluation. This task is approached by tackling several sources of inefficiency in modern architectures and application mapping techniques alike. A well-known method to improve efficiency is through specialisation, e.g., in the form of application-specific instruction-set processors (ASIPs) or even application-specific integrated circuits (ASICs). Consensus is however that specialisation reduces flexibility of the system, especially when specialisation is taken to a point where applications outside of the target set are no longer supported. Loss of flexibility is generally undesirable, in particular for systems that are to execute ANNs since the underlying algorithms mutate heavily in the context of intense research in this field. A proper quantitative definition of system flexibility does not exist however, making it difficult to objectively assess the flexibility of a compute system. To overcome this gap in compute theory, this thesis also includes an attempt to systematically provide both a qualitative and quantitative definition of compute system flexibility.

Just like Archimedes recognized the need for a solid foundation to place his lever upon, this thesis builds upon the work of those who came before. Starting from state of the art architectures and techniques, contributions are made to improve compute efficiency and data efficiency as further elaborated in section 1.1.1 and section 1.1.2 respectively. Finally, an attempt is made to solidify the theoretical base of the field by introducing a quantitative metric for compute flexibility as discussed in section 1.1.3. The contributions made in this thesis are summarized in section 1.2, and section 1.3 provides an overview of the remainder of this thesis.

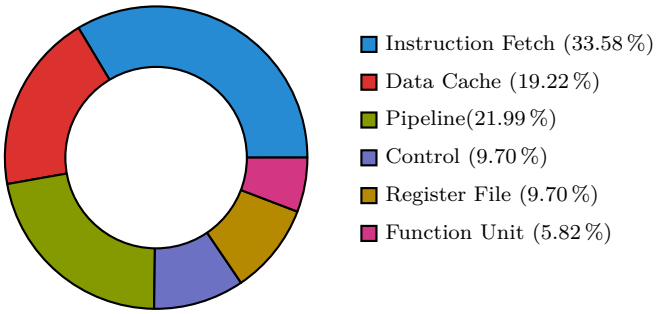


Figure 1.3: Energy distribution in a four core chip multiprocessor based on data by Hameed et al. [50, Table 3].

1.1 Problem Statement

To advance the complexity of algorithms that can be executed cost-effectively on flexible compute systems, it is necessary to improve the energy efficiency of these systems. This holds especially for artificial neural networks because of their compute intensive nature, and requirement for flexible systems to cope with the highly volatile context in which the machine learning field currently finds itself. The objective of the work described in this thesis is to *improve the efficiency of compute systems in general, but with a focus on artificial neural networks in particular.*

To improve energy efficiency of compute systems, first the sources of inefficiency should be understood. An important benchmark in this area is the paper titled "Understanding Sources of Inefficiency in General-Purpose Chips" by Hameed et al. [50]. Based on Table 3 of this paper, the pie-chart in figure 1.3 can be constructed which contains a breakdown of the energy usage in a baseline quad core multiprocessor. With the function unit, the logic which performs the actual computation, only contributing 5.82 % of the total energy consumption, it is evident there is much to be gained by reducing the other sources of energy consumption.

The efforts towards improving the energy efficiency of flexible compute systems captured in this thesis are logically divided into three distinct parts:

1. *Compute Efficiency* - The first part addresses compute efficiency, which deals with improving the general energy efficiency of computations. This

topic is investigated both at the logic block level, as well as an architectural viewpoint concerning instruction flow (section 1.1.1).

2. *Data Efficiency* - The second part covers scheduling techniques that focus on reducing data movement, which, due to the gap between logic and memory scaling, can have a significant impact on the overall energy efficiency of a system (section 1.1.2).

3. *Compute System Flexibility* - The third and final part of this thesis treats the concept of compute system flexibility, and how this property is affected by various architectural choices, enabling designers to make conscious, balanced trade-offs when constructing a compute system (section 1.1.3).

1.1.1 Compute Efficiency

Compute efficiency refers to the energy efficiency with which computations are executed on a system. This includes both the energy spent on the actual computation of the result of an operation, as well as the energy used for the supporting control logic. Or, in terms of figure 1.3, everything except the data cache which is explicitly covered separately in the second part of this thesis as described in section 1.1.2.

From the conceptual Turing machine to modern-day specialised processors, a common denominator between all flexible computers is the fundamental cycle of instruction selection and evaluation, which typically both consume energy. It can be seen in figure 1.3 that in modern multi-core processors the energy consumed by the instruction fetch dominates, and is almost six times higher than the energy used by the function unit. A well-know mitigation to this problem is to employ a single instruction multiple data (SIMD) machine [2, 56, 93, 116, 132, 173], which fetches one instruction and applies it in parallel to many elements. Apart from amortising the instruction fetch and control energy, this scheme also increases parallelism. A major advantage of this increased execution parallelism is that the same throughput can be achieved at a lower clock frequency, potentially enabling supply voltage scaling further improving energy efficiency. When an application contains sufficient data-level parallelism (DLP), a wide-SIMD machine can greatly improve the energy efficiency compared to a multi-core machine without vector extensions.

With instruction fetch and control amortised over many parallel operations in SIMD machines, the energy consumption of the datapath and register file become dominant as will be demonstrated in chapter 2. The datapath burns a large

amount of energy by moving values through a pipeline every clock cycle, even when certain control bits or results may not be relevant for every instruction. On top of this, a typical reduced instruction set computer (RISC) machine instruction accesses the register file with two reads and one write per evaluation. These sources of energy consumption in the datapath remain the same across all processing lanes in an SIMD machine, and as such are not amortised. Resolving these two energy bottlenecks is one of the challenges that need to be tackled to increase the compute efficiency of SIMD machines.

Apart from the register file accesses and pipeline registers, wide-SIMD machines suffer from an issue specific to this architecture; As long as the operations that are executing in parallel are truly independent, there is no limit to the scaling of an SIMD machine. When communication between the lanes is required, however, this may become a bottleneck in scaling. A dedicated communication network that links all SIMD lanes directly is expensive in area and does not scale. For a truly scalable solution, the communication network should only rely on local connections. When not used in a smart manner, only local links may slow down communication significantly, up to the point that it kills all performance and energy gains from scaling to a wider SIMD machine. An important case where SIMD lanes need to communicate is the reduction operation seen in convolutional neural networks (CNNs), at the time of writing the most popular subclass of artificial neural networks. Rather than adding a costly communication network or dedicated reduction logic, the challenge to be solved is maintaining high performance using a communication network that only depends on local links.

Assuming the energy consumption of the pipeline and register file can be reduced, the compute itself starts to become relevant for further optimisation. As will be detailed later in chapter 4, there is a general inefficiency in standard functional units which makes that a relatively simple calculation such as 1×-2 consumes about the same energy as 149132×24132, i.e., the energy consumption barely depends on the operands. Intuitively it should be possible to perform the former operation with less energy than the latter. This is especially interesting for multiplication, which is a dominant part of the operation energy in many applications including CNNs. The last challenge regarding compute efficiency that is addresses in this thesis, is the dynamic exploitation of the lower complexity of certain operations based on the data-width of their operands.

1.1.2 Data Efficiency

With the compute efficiency addressed in the previous part, what remains in terms of inefficiency sources in figure 1.3 is the data-cache and memory accesses in general. Note that the energy cost of accessing a large main memory is not included in the numbers by Hameed et al. [50] on which figure 1.3 is based. This will add a large additional source of inefficiency to the overall picture, since compared to an arithmetic logic unit (ALU) operation, accessing an on-chip static random-access memory (SRAM) requires about 5× the energy, and going to external dynamic random-access memory (DRAM) about 200× [124]. To make matters worse, this phenomenon which is commonly referred to as the memory wall [177] will only aggravate with further technology scaling.

A major challenge is to decrease the energy consumed by loading, storing, and moving data. The common approach is to employ multi-level memories, which exploit properties of various types of physical memories. Generally, large memories which allow dense storage require more energy and time to access than small memories built in standard logic. By creating a memory system of several layers, it is generally possible to perform the majority of accesses on the fast, cheaper to access, low-level memories. Such a hierarchical system depends on the concept of reuse, where a single data element is used in multiple computations. Without reuse, the smaller, faster memories would not provide any advantage over going to main memory directly.

Fortunately many applications exhibit data reuse. The challenge, however, is to expose and exploit this reuse optimally. Often there is freedom in selecting the order of execution of operations in an application. Different execution schedules will result in different reuse distances, i.e., the number of unique accesses between the uses of a specific data element. The selection of the right schedule can improve the reuse captured in the lower memory levels, and as such reduces the energy consumption of the overall application. The number of data uses is determined by the application, but within the given data dependencies they can be scheduled in various orders. For deep learning applications in particular, with their associative multiply accumulate (MAC) operations, this scheduling space is vast. This makes finding efficient schedules a non-trivial task which needs to be solved to improve the energy efficiency of ANNs.

1.1.3 Flexibility

The term flexibility and claims regarding its relation to other architectural properties such as energy efficiency are thrown around a lot [75, 76, 105, 126, 171]. In particular there appears to be some consensus on how higher flexibility is related to supporting a wider set of applications, or changing applications, but also with a lower energy efficiency. However, flexibility as a property is ill-defined. If for example the set of supported applications is taken as a measure of flexibility, then any Turing-complete system will be maximally flexible by definition. It does not relate this property of 'support' in any form with how well it is supported, or how well a set of, potentially changing, applications is supported, and therefore does not provide a solid way to compare system flexibility.

The lack of an objective, quantifiable definition of flexibility hampers discussion and the appropriate balancing of computer architectures. To design a system that is 'sufficiently flexible', first flexibility has to be defined in a systematic manner that finds a following within the computer architecture community. This challenge is larger than this single thesis, but an attempt at such a definition is made nonetheless with the hope that it will start a broader discussion within the community. Eventually an established definition will help designers better understand the machines they architect, and enable an informed trade-off between flexibility and energy efficiency, assuming such a relation in fact does exist.

1.2 Contributions

The contributions of this thesis include:

- A configurable wide-SIMD architecture with minimal, but sufficient, interconnect and explicit datapath to reduce register file energy consumption. Extensive experimental results show that an instance with 128 processing elements is capable of speeding up an application by 206× compared to a reference RISC processor while reducing energy consumption by 48.3 % on average. 27 % of these energy savings can be attributed to the explicit datapath, which is particularly effective in wide-SIMD architectures (chapter 2).
- Reduction algorithms that require only minimal interconnect on SIMD machines. These algorithms provide an alternative to dedicated reduction hardware such as applied in an adder tree. Results show that at a perfor-

mance penalty of 7.1 % the algorithms save 6.8 % in logic area in a 128 wide-SIMD machine compared to a solution employing an adder tree, while maintaining generality of the reduction operator (chapter 3).

- Several data-width aware multiplier designs which improve the energy efficiency of traditional hardware by dynamically exploiting the effective data-width of the operands. In particular a sign-magnitude based design is presented which, even when transparently integrated in two's complement datapath, is shown to improve energy efficiency by $1.38 - 2.25\times$ (chapter 4).
- A fast mathematical model of the scheduling space of convolutional neural networks and accompanying open source tool for fast automated design space exploration. The model estimates the memory and compute requirements of a wide range of neural network schedules, and provides designers with a Pareto graph [113] to trade-off internal buffer size versus the number of required external accesses and the number of MAC operations. Experimental results show improvements in energy efficiency of an order of magnitude can be achieved (chapter 5).
- In lack of a proper definition of compute flexibility, a systematic attempt is made to provide both a qualitative an quantitative metric for flexibility. The proposed metric is extracted for 25 platforms with varying compute architectures, and compared to common notions of flexibility found in literature. The hope is that this chapter sparks a broader discussion in the computer architecture community, and leads to a single, commonly accepted metric for flexibility (chapter 6).

1.3 Thesis Overview

A graphical overview of the thesis structure is provided in figure 1.4. As can be seen, the remainder of this thesis is organised as follows. In chapter 2 a configurable, wide-SIMD architecture is introduced, which employs explicit datapath techniques to improve its compute efficiency. Chapter 3 discusses how the minimal, scalable interconnect of the proposed SIMD can be used efficiently to perform reduction without the need for dedicated reduction hardware, achieving good performance without specialisation for this important operator. Finally, chapter 4 concludes the work on improving compute efficiency by investigating how to dynamically exploit the data width of operands to reduce energy consumption. Next, chapter 5 presents mathematical models to enable fast schedule

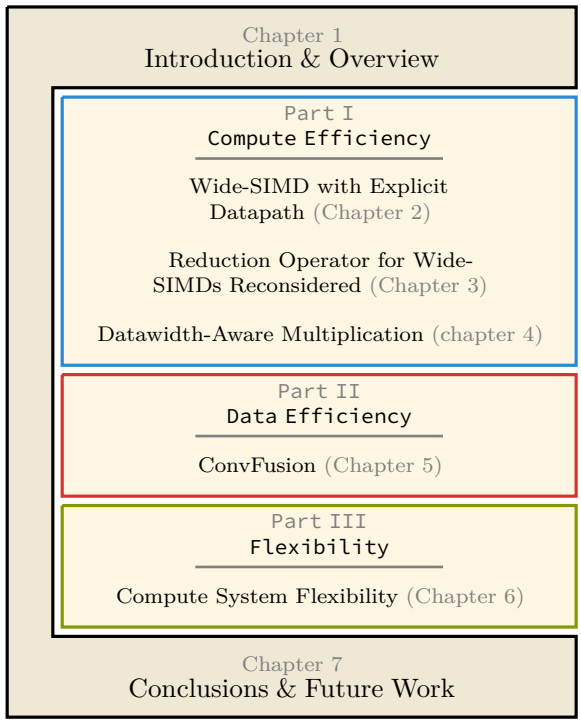
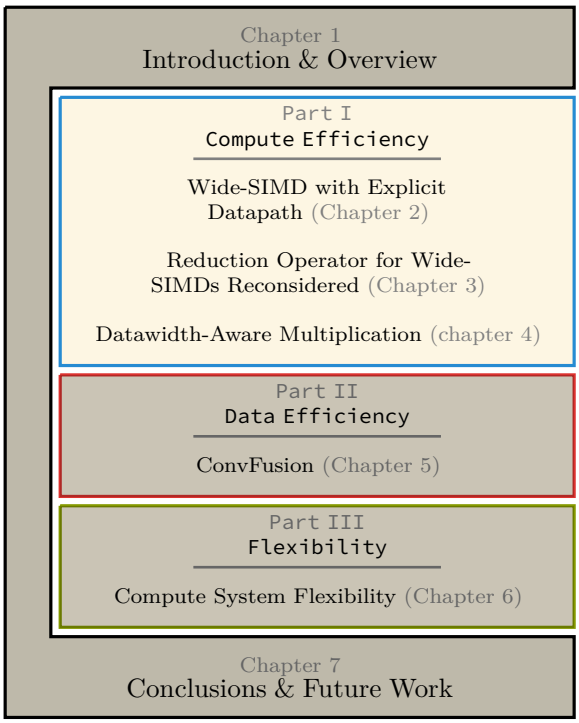


Figure 1.4: Thesis structure.

space exploration for CNNs. The last part, defining compute system flexibility, is covered in chapter 6. Where each individual chapter is ended with self-contained conclusions, chapter 7 finally concludes the overall thesis.

Part I
Compute Efficiency



Chapter 2

Wide-SIMD with Explicit Datapath

This chapter is based on the work published in "SIMD made explicit" [159] and "A Low-Energy Wide SIMD Architecture with Explicit Datapath" [157].

From the conceptual Turing machine to modern-day specialised processors, a common denominator between all computers is the fundamental cycle of instruction selection and evaluation, both which consume energy. A classic approach to reduce the energy required for the instruction selection is to evaluate a *single instruction* for *multiple data* elements (SIMD), effectively amortizing the cost of the instruction selection. This concept is taken to the extreme in the configurable wide single instruction multiple data (SIMD) machine proposed in this chapter, and, crucially, combined with an explicit datapath to reduce the energy consumption by avoiding accesses to register files in each lane of the SIMD machine. Extensive experimental results show that the proposed architecture is efficient and scalable in terms of area, performance, and energy. In particular, a 128 processing element (PE) instance of the proposed architecture is shown to achieve an average speedup of 206×, while reducing the energy dissipation by 48.3 % on average and up to 94 % in the best case compared to a reduced instruction set computer (RISC) processor. The results show that explicit datapaths are particularly effective in wide SIMD machines. Compared to the corresponding SIMD architecture without explicit bypassing, an average of 64 % of all register file accesses is avoided by the 128-PE instance, improving the compute energy efficiency of the baseline machine by 27 % on average, and up to 43.0 % in the best case.

2.1 Introduction

When high compute performance is required under a limited energy budget, a solution may be found in wide-single instruction multiple data (SIMD) architectures [116, 2, 172, 74]. In SIMD machines, a single instruction operates on multiple data elements in parallel. This enables SIMD machines to exploit the data-level parallelism (DLP) present in an application. Because multiple operations are carried out simultaneously, high computational throughput can be delivered at a very low clock frequency and thus low voltage, thereby greatly improving energy efficiency [56, 118]. Another important energy efficiency enhancing feature of wide-SIMDs is the sharing of a significant portion of the control and datapath between multiple processing elements (PEs).

The particular wide-SIMD proposed in this chapter consists of a control processor (CP) and a configurable number of PEs. Since the PEs execute the same instruction in each cycle, the instruction fetch (IF) and instruction decode (ID) hardware can be shared between all PEs. The energy usage of the instruction memory (IMEM), a significant part of the total energy usage in a single core processor [50], is amortized over multiple PEs. Furthermore the control flow is shared, as it is managed by the CP. For a wide-SIMD with hundreds of PEs, the energy used by these shared parts, i.e., IF, ID, and control flow, becomes almost negligible on the overall energy usage. The largest amount of energy is dissipated in the PE datapath [56, 118] which performs most of the useful computations, resulting in a high energy efficiency. Because of this, techniques that improve the energy efficiency of the PE datapath are particularly effective in wide-SIMDs.

Since the energy usage of a wide-SIMD is dominated by PEs, and the register file (RF) is one of the main contributors to the PE's energy dissipation, as will be further elaborated in section 2.4, reducing the energy dissipation of the RF has a large impact on the overall energy usage of wide-SIMD architectures. It is therefore imperative to reduce the energy usage of the RF in order to improve the overall energy efficiency of a wide-SIMD architecture.

Explicit bypassing is a well-known technique that can reduce the energy usage of an RF [57, 45, 39, 179]. Traditional pipelined architectures usually have hardware bypassing[1] mechanisms to mitigate/remove the penalty of read-after-write hazards. This bypassing is dynamically performed by hardware, and thus completely transparent to software. In this thesis, this hardware controlled form of bypassing is referred to as *implicit* bypassing. *Explicit* bypassing, on the

[1] Also known as 'forwarding'

contrary, directly controls the bypassing network in software. Explicit bypassing has the potential to greatly reduce the number of accesses to the RF, resulting in a higher energy efficiency [57, 45, 39].

In this chapter a programmable, highly energy efficient, configurable wide-SIMD architecture that exploits the explicit datapath concept is proposed. A complementary tool flow composed of compiler, simulator, and hardware description language (HDL) generator, is also developed for the proposed architecture. To demonstrate the efficiency of the proposed architecture, and the effectiveness of explicit bypassing in wide-SIMDs, multiple instantiations of the proposed wide-SIMD architecture and its implicitly bypassed counterpart, as well as a baseline reduced instruction set computer (RISC), are synthesized with a commercial 40 nm low-power library. Eleven representative kernels are chosen to examine the proposed architecture. These kernels contain different types of communication and memory access patterns, namely *point-to-point, neighbourhood-to-point, global-to-point, and global-to-global* (described in section 2.3), which represent a wide range of applications. Additionally, fast focus on structures (FFOS) [58, 53], a complete industrial computer vision application, is used to evaluate the proposed architecture.

The experimental results show that in a 128-PE instance of this SIMD processor is able to achieve an average speedup of 206× and reduces the total energy dissipation by 48.3 % on average and up to 94 %, compared to the baseline RISC machine at the same clock speed and supply voltage. Compared to the corresponding SIMD architecture with implicit bypassing, an average of 64 % of all RF accesses is avoided by the 128-PE, explicitly bypassed SIMD. This results in an average reduction of 27.5 % on the overall energy consumption.

The key contributions reflected in this chapter are:

- Proposal of a configurable, highly energy efficient, wide-SIMD processor architecture with explicit datapath.
- Systematic evaluation of this architecture in terms of silicon area, performance, energy, and scalability using benchmarks with different communication and memory access patterns.
- Analysis of an industrial case study demonstrating the effectiveness of the proposed architecture.

The remainder of this chapter is organised as follows: Section 2.2 introduces the proposed architecture and elaborates the differences between explicit and

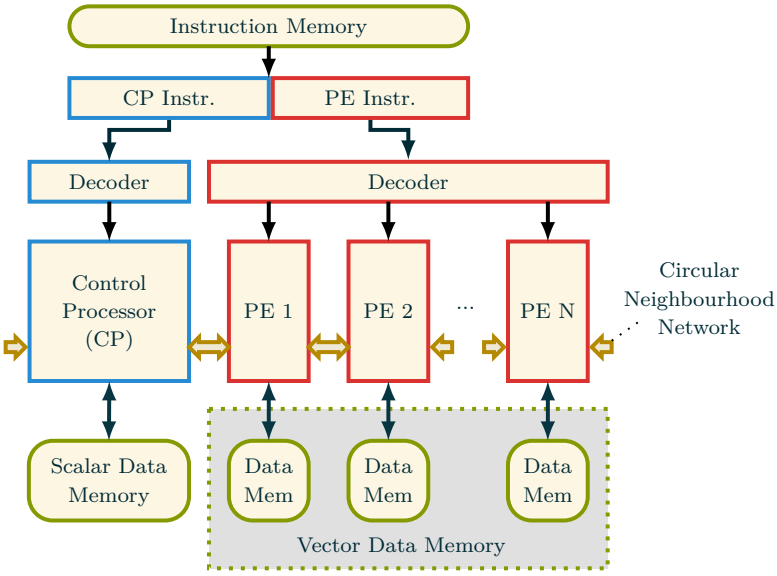


Figure 2.1: High-level structure of the proposed wide-SIMD. A single control processor runs in lock-step with an array of processing elements.

implicit bypassing. Section 2.3 describes the experimental setup and benchmarks. Experimental results that show the effectiveness of the proposed architecture are provided in section 2.4. Related work is discussed in section 2.5. Finally, section 2.6 concludes this chapter.

2.2 Proposed Wide SIMD Architecture

As discussed in section 2.1, a wide-SIMD architecture is an excellent candidate for an energy efficient compute platform. In a wide-SIMD, many PEs execute the same instruction. Hence IF and ID overhead are amortized over all PEs. Additionally, wide-SIMDs are able to meet high compute demands at low clock frequencies and voltages, further improving their energy efficiency.

The proposed processor architecture as shown in figure 2.1 consists of two parts; a CP, and a wide one-dimensional array of PEs which runs in lock-step with the CP. This essentially makes up a very long instruction word (VLIW) processor with one scalar issue slot for the CP, and one (wide) vector issue slot for the PE

array. DLP is exploited in the PE array, while instruction-level parallelism (ILP) is exploited through issuing scalar and vector operations simultaneously. This is particularly effective in an SIMD context, where without the exploitation of ILP by the CP, a single scalar instruction would require the whole PE array to execute this one scalar instruction. By executing scalar instructions in parallel to vector instructions, an ideal speedup of two can be gained over any speedup already obtained by exploiting DLP in the PE array. The CP and PE array operate in lock-step, such that the control flow for both entities is uniform. This enables the CP to handle the control flow, while the PE array processes data in parallel.

In the proposed architecture each PE has a private data memory (DMEM) with its own address generator, i.e., per-PE local addressing. Despite the higher area and logic cost compared to global addressing for all PEs, per-PE addressing results in much better programmability [60]. Many applications, such as histogram calculation and Hough transform can benefit from this independent address generation [59, 60], offsetting the overhead compared to globally addressed memory. In line with the PEs, the CP also has a private scalar DMEM.

The IMEM is shared between the CP and the PE array. Each instruction stored in the IMEM contains a pair of operations, i.e., one CP (scalar) operation and one PE-array (vector) operation. The vector part of the instruction is fetched from the shared IMEM, partially decoded in the shared ID stage, and broadcast to all PEs. The broadcasting of these signals across the chip could in a silicon implementation negatively affect the maximum achievable clock frequency. Synthesis results show that this is not an issue for the target frequency of 100 MHz. In case a higher clock frequency is required, a solution may be found in pipelining these long wires at the penalty of increased branch delay slots. This in turn may be alleviated by employing an accurate branch predictor, of which the energy cost will anyway be amortised over all PEs.

The instruction set architecture (ISA) of both the CP and the PEs is based on a 24-bit RISC-like ISA, similar to the one used by She et al. [133], but with two extra bits for neighbourhood communication as detailed in section 2.2.2, and two extra bits for predication as detailed in section 2.2.4. The instruction width for both the CP and the PEs is therefore 28-bit instead of the original 26-bit [133]. An overview of the ISA is provided in appendix A.

2.2.1 Datapath

The proposed wide-SIMD framework supports different datapath configurations (section 2.2.5), e.g., a 4- or 5-stage pipeline. For brevity, this chapter describes the 4-stage RISC-like datapath version only.

One of the properties of SIMDs is that the IF and part of the ID logic are shared among all PEs. The remaining parts of the decoding logic, such as RF accesses and operand selection, are performed locally in each PE. The execution stage (EX) contains an arithmetic logic unit (ALU), a multiplier unit (MUL), and a load store unit (LSU). The write back stage (WB) commits the results to the RF if necessary. To optimize the datapath for low energy usage, each functional unit (FU) in the execution stage has its own input registers. This is to isolate/clock-gate FUs, such that an FU does not dissipate dynamic power if it is not the target FU of the current operation. Another important reason to introduce input registers for each FU is to extend the available time of FU outputs in the bypassing network which can improve the efficiency of the explicit datapath Before moving to the main focus of this chapter, i.e., the *explicit datapath*, the next section first presents a conventional *implicit bypassing datapath*.

Implicit Bypassing Datapath

Conventional pipelined architectures typically have an implicit bypassing mechanism to mitigate the penalty of read-after-write hazards. This bypassing is transparent to software. In an implicitly bypassing datapath, hardware keeps track of all the uncommitted results in the pipeline and determines whether a bypass is required. Figure 2.2 depicts a 4-stage datapath with implicit bypassing. In this datapath there are two bypass sources, i.e., the result of the EX stage and the output of the WB stage.

Despite being widely used, implicit bypassing has two major disadvantages with respect to energy efficiency:

1. *Speculative Reads*
   The detection of bypass situations is typically performed in the ID stage, in parallel with the operand fetching from the RF. Therefore, the operands specified by an instruction are usually speculatively fetched from the RF. If a bypass is required, the operand fetched from the RF is invalid and will be discarded. The energy used to fetch the discarded operand from the RF is thus wasted.

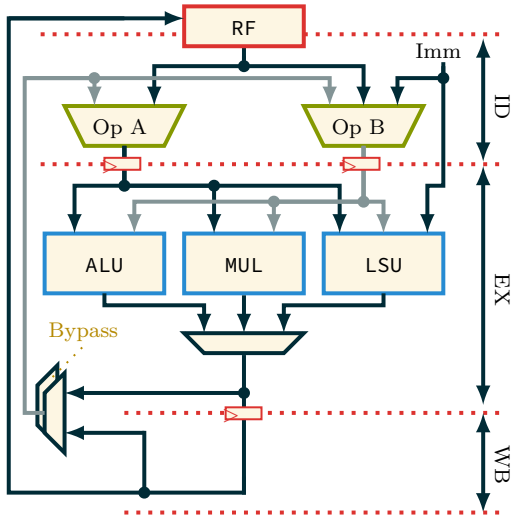


Figure 2.2: Datapath with implicit bypassing. Hazard detection logic (not shown) dynamically controls the bypass mux to resolve pipeline hazards.

2. *Speculative Writes*
   In programs with short-lived values, computed results may be consumed through a bypass. If all uses of this result are bypassed, writing this result back to the RF is not necessary. In an implicitly bypassing datapath, however, the hardware cannot determine whether a variable is going to be referenced in the future or not. Therefore, the *dead* result is always written back to the RF, speculating it will be used in the future. Writing a variable to the RF that is never referenced again wastes energy.

Speculative reads are caused by the lack of time to detect such bypass situations dynamically before the RF is accessed. These reads may be avoided by the addition of an extra pipeline stage, but as will be detailed in the next section an explicitly bypassing datapath is a far more elegant solution. Speculative writes find their origin in the fact that hardware has no liveness information of the variables, since it can only observe those variables that are currently in the pipeline.

Explicit Bypassing Datapath

Explicit bypassing is a technique that can be used to reduce the energy usage of an RF [57, 45, 39, 179, 54]. The key idea of explicit datapath architectures is to expose more details of the datapath to the compiler, thereby enabling fine-grained control over the datapath in software.

The disadvantages of implicitly bypassed datapaths originate from the lack of liveness information and limited time to detect bypass situations. If the compiler is given information on the architecture of the datapath, however, the situations that require bypassing can be easily detected at compile time. By keeping track of where all the variables are in the pipeline, the compiler can determine what bypasses are required. Furthermore, the compiler has a full view of the liveness of all the variables. Given that the compiler can detect and control what values/operands to bypass in the pipeline, it can also determine whether a variable is dead before it is committed to the RF. In that case it can prevent the speculative write that implicit bypassing needs to perform. In this way the disadvantages of implicit bypassing can be overcome by shifting the control of the bypass logic from hardware to the compiler.

To enable the compiler to control bypassing, the actions required to perform a bypass need to be encoded in the ISA. In the proposed datapath this is achieved by mapping bypass sources to the register address space. When a bypass is needed from a certain bypass source, the compiler inserts the address associated with that source into the corresponding operand field. Furthermore, if a variable does not need to be written to the register file, the compiler will replace the destination address with `r0`, or its notational alias `--`, which is used for readability in the remainder of this thesis. Register `r0` always reads zero, and when the hardware encounters it as a write destination no actual write is performed.

Figure 2.3 depicts the explicit datapath of the proposed architecture. There are four bypass sources, namely the ALU output, MUL output, LSU output, and WB output. These sources are mapped to the end of the 5-bit register space. To enhance readability of code, an alias is defined for each of the bypasses. `ALU` indicates the result of the ALU, `MUL` of the multiplier, `LSU` of the load store unit, and `WB` indicates the writeback stage bypass source.

Compared to an explicit datapath implementation with less bypass sources, such as the work of Yan et al. [179], multiple bypass sources can further reduce the traffic to/from the RF, as will be shown in an example later in this section. In

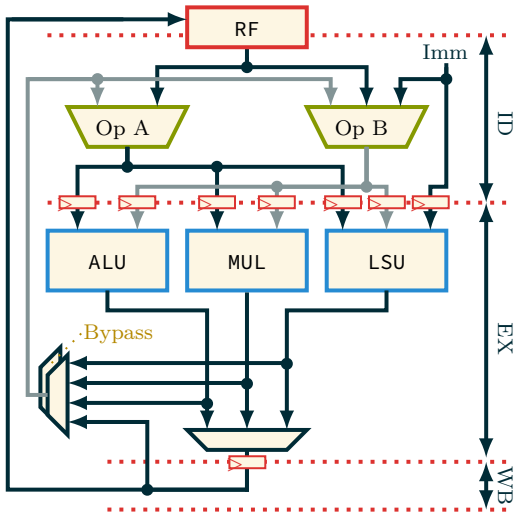


Figure 2.3: Datapath with explicit bypassing. Software explicitly controls the bypass mux, which compared to the implicit datapath can select from more bypass sources.

the proposed architecture, since the bypass sources are mapped to the end of the 5-bit register space and RF-write removal is achieved by *writing* to a virtual register `r0`, no extra bit(s) need to be added to the instruction format. The cost of this encoding is the loss of four entries in the RF.

Another low-energy optimization featured in the proposed architecture are the private input registers for each FU in the execution stage. These input registers are only updated when the FU is active. Not only does this prevent unnecessary toggling in these units, but it also preserves the output of the FU when not in use. This extends the time results remain in the pipeline, increasing bypass opportunities. The remainder of this section discusses these features in more detail.

For an example of explicit bypassing, consider code 2.1. The first operand (`r1`) of the **`add`** instruction on line 2 needs to be bypassed from the output of the multiplier (`MUL`) to the input of the ALU. The compiler will in this case insert the register address associated with the `MUL` bypass source as shown in code 2.2. The hardware will in this case fetch the operand from the output of the multiplier. Because the bypass situation can be directly extracted from the instruction, the RF does not need to be accessed. Additionally the result of the **`mul`** instruction

```
1  mul r1,   src1, src2
2  add dest, r1,   src2
```

Code 2.1: Simple bypassing situation.

```
1  mul --,    src1, src2
2  add dest, MUL,  src2
```

Code 2.2: Explicitly bypassed version of code 2.1

on line 1 does not need to be written to the RF, because after the bypass the variable is dead. The compiler can analyse this and insert the `--` address as the destination to avoid the write.

The explicit datapath increases the bypass opportunities by exposing more bypass sources compared to the implicitly bypassed pipeline. Essentially, the bypass source of the execution stage in figure 2.2 is split into three separate sources, ALU, MUL and LSU, as shown in figure 2.3. The input operands of these units are furthermore only updated when a unit needs to be active. This avoids unnecessary toggling of inactive units. It also ensures the output of a unit remains unchanged until the next operation that uses it. As the input operands do not change, the output of the logic holds the output value of the previous operation. This enables bypassing of these outputs as long as the associated unit is not used, increasing bypass opportunities.

To demonstrate the effect of adding more bypass sources, consider code 2.3, which would only have one bypass with the bypass sources from figure 2.2. By adding the bypass sources as depicted in figure 2.3 this code is transformed into code 2.4. The difference is significant, from only one read avoided by bypassing in code 2.3, the added bypass sources avoid another six reads, yielding a total of seven avoided reads. On top of this the additional sources avoid four writes to the RF, halving the number of writes.

The drawback of adding extra bypass sources is the additional locations they take up in the RF address space. The implicit datapath has 32 registers, while the explicit datapath has only 28, because four locations are used to address the bypass sources. However, as has already been shown in various related work, explicit bypassing can largely mitigate the increased RF pressure as no registers need to be reserved for variables that are consumed through bypassing [39, 31, 146, 134, 179]. This can also be observed when comparing code 2.3 and code 2.4.

```
1  lw   r2,   r1,   0
2  add  r1,   r1,   1
3  mul  r3,   WB,   4  // r2 is bypassed with WB
4  lw   r4,   r2,   0
5  sw   r0,   r1,   1
6  add  r5,   r3,   r1
7  mul  r6,   r3,   r4
```

Code 2.3: Code snippet that has only one bypass when there is only a single bypass source for the EX stage as described in figure 2.2

```
1  lw   --,   r1,   0
2  add  --,   r1,   1
3  mul  --,   LSU,  4
4  lw   --,   LSU,  0
5  sw   r0,  ALU,  1
6  add  r5,  MUL,  ALU
7  mul  r6,  MUL,  LSU
```

Code 2.4: Same code as in code 2.3, but now bypassed with the additional bypass sources as in figure 2.3

In the original situation six registers are used. After adding the additional bypass sources only three general registers are required.

A long-standing issue of explicit datapath is the much larger state that needs to be saved for interrupts and context switching [29]. When the proposed architecture is used in its intended setting as an accelerator it may not be required to support interrupts. When interrupts are required, a possible solution is to use a scan chain to stream out the current state without a large hardware overhead.

2.2.2 Interconnect

One of the main challenges of wide-SIMDs is the interconnect between PEs. Many practical applications require some form of communication between the processing units. For example to access input data or processed results stored in the DMEM of a different PE, or for synchronization between the PEs. From an application point of view, the most convenient form of interconnect is a fully connected network. Such networks scale extremely poorly in hardware, however, due to their logic complexity and long wires. A number of interesting solutions have been proposed for wide-SIMD, such as a modified crossbar [125], an X-RAM swizzle network [131], and dynamic communication for SIMDs [35]. Although

these networks mitigate the problem to a degree, certain shortcomings remain. Some are optimized for only a few specific applications. DC-SIMD [35], e.g., is mainly tuned for kernels such as lens distortion correction. Other solutions require a large amount of additional hardware [125, 131]. In all cases scalability is limited, which is undesirable for application in a generic wide-SIMD.

Circular Neighbourhood Communication Network

In the proposed architecture, a circular neighbourhood network is employed which is highly similar to the neighbourhood network of the Xetal SIMD processor [2]. The main difference is that the neighbourhood network in the Xetal processor has constrained access to data of its direct neighbours, e.g., a PE can access data in the data memory of its direct neighbour, but all PEs have to access with the same address/index to its neighbour's memory. This is because the Xetal processor does not support per PE addressing and it also lacks predication support. The circular neighbourhood network in the proposed architecture has more flexible access to data of its direct neighbours as both per PE addressing and predication/execution guarding are supported.

In a circular neighbourhood network, the units are logically connected in a ring where each unit can communicate with its left and right neighbours as shown in figure 2.4. Such a network requires exclusively short wires, allowing virtually unlimited scaling. This is backed up by the results presented in section 2.4.1 and section 2.4.2. To support neighbourhood communication, two additional bits are added to the ISA, indicating from which source (i.e., left neighbour, right neighbour, or 'self') the operand should be read.

The neighbourhood network performs extremely well for local, short distance communication as encountered in many typical kernels on SIMDs, such as box filters, convolution, and motion estimation. In the target application domains of wide-SIMDs, such as image and video processing, many kernels show high locality in their communication, making a neighbourhood network highly suited.

When two PEs which are not direct neighbours need to communicate, data must be moved through all units in between. This is especially costly if long-distance communication is required extensively. Unfortunately, these kinds of applications are by no means rare. Examples of kernels that require long-distance communication between PEs are: *partial histogram merging*, *row projection* (sub kernels in the industrial benchmark application described in section 2.3.2), *max vector element*, and *sum of vector elements* (categorized as *global-to-point* kernels

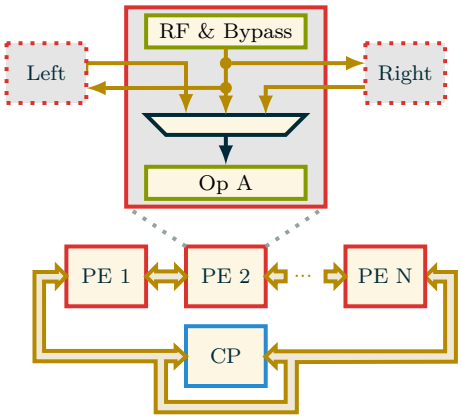


Figure 2.4: Circular neighbourhood communication network. Processing elements are connected to their direct neighbourhoods in a ring which optionally can include the control processor.

in section 2.3.2). To efficiently handle these kernels on a wide-SIMD with only a circular neighbourhood network, two novel reduction algorithms are introduced in chapter 3, *pipelined reduction* and *diagonal access reduction*, which do not rely on complex communication networks or any dedicated hardware. The key idea of both approaches is to utilize inter-vector parallelism instead of intra-vector parallelism. Experimental results show that the performance of the proposed algorithms is comparable to dedicated reduction hardware.

As can be seen in figure 2.4, the CP takes a special place in the circular neighbourhood network to facilitate both communication among PEs and between PE-array and CP. By dynamically programming the border PEs of the PE-array, it is possible to eliminate the CP from the circular network. Furthermore the border PEs can be programmed to not read from each other, but reading a fixed value instead. This is useful for automatic insertion of image borders for example.

2.2.3 CP Broadcast

In addition to the circular neighbourhood network described in the previous section, direct broadcasting from the CP to all the PEs in the PE-array is also supported. Some applications require that the exact same data is sent from the CP to all PEs. For example a centrally computed threshold. Although it

would be possible to use the neighbourhood communication network for this purpose, the total number of required hops would make this an highly inefficient approach. Instead the CP is given the ability to broadcast data that can be read by PEs in a manner similar to reading neighbouring operands. This mechanism does introduce long wires to the design again, which negatively affects scalability. However, these wires can be pipelined to not harm the achievable clock frequency at the cost of increased broadcast delay. In particular because wide-SIMDs target low frequencies, the number of required pipeline stages is likely low as well. Also strategic placement of the CP close to the centre of the die will enable a reasonably efficient tree-like broadcast structure.

2.2.4 Predication

Strictly speaking the SIMD paradigm applies exactly the same instruction to multiple data elements in parallel. This is however not always desired. Sometimes it is required by the program flow that some PEs do not perform a certain operation for example. To facilitate this diverseness in program flow, each instruction is prefixed with two predication bits. Each PE can set and clear two predication flags with compare operations in the ALU. An instruction will only be executed on a particular PE if its corresponding flags are set, otherwise a no-operation (NOP) is inserted. This enables for example the implementation of `if-then-else` constructs on the PEs. Each PE can determine whether it needs to execute the `then` or `else` code block.

Note that when predicates are used to support diverging program flows, care has to be taken in context of the explicit datapath when divergent execution flows join. After all, in the explicit bypassing code the state of the pipeline is presumed to be known, and equal for all active compute elements. When execution paths between PEs first diverges through the use of predicates, the state is still equal for all paths. During the execution of the different paths the executed instructions may be different between the divergent paths however. When two divergent paths eventually converge, the state of the pipelines of the PEs of each branch may not be consistent. To solve this, it may be necessary to perform additional operations to equalize the different pipeline states. The straightforward, but also costly approach is to ensure that before any join in the control flow the pipeline states are flushed by inserting dummy instructions ensuring all state inside the pipelines is committed to memory. This overhead may be prohibitive to performance when it occurs in the inner loops of critical code. Typically we find however that this drastic measure is not required, and

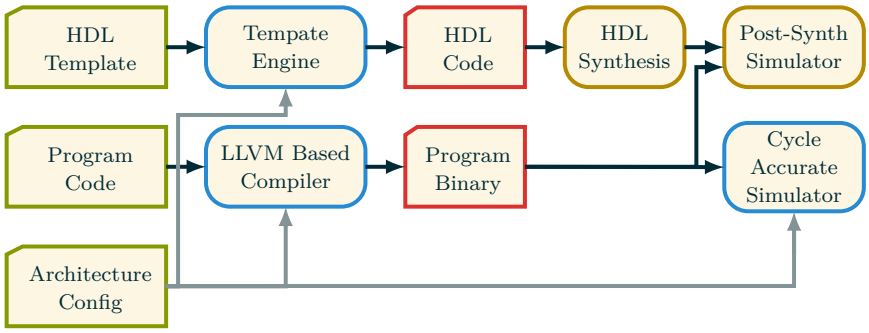


Figure 2.5: The framework capable of generating instances of the proposed wide-SIMD architecture based on a configuration file. A single architecture configuration file controls hardware & and code generation, and the behaviour of a cycle-accurate simulator.

imbalanced paths can aligned using only one or two additional operations. In the code of the benchmarks further detailed in section 2.3.2, only one benchmark required additional instructions to align diverging paths, and this was in a non-critical part of the code resulting in negligible performance loss. Note that although this is true for the proposed 4-stage datapath, it likely does not hold generically for more deeply pipelined architectures.

2.2.5 Configurable Framework

To enable fast design space exploration and to tailor the proposed architecture for different target applications, a design framework is developed to easily generate different instantiations of the architecture with, for example, varying number of PEs, 4-stage or 5-stage datapath, explicit or implicit bypassing, as well as different datapath widths. The complete set of parameters varied in this chapter is presented in table 2.1.

Figure 2.5 shows the high-level diagram of the developed framework. The architecture configuration file is a human readable JSON file that specifies an instance of the wide-SIMD architecture. The hardware toolflow is visualized at the top half of figure 2.5. An HDL template is combined with the architecture configuration file to generate corresponding HDL code of the instance specified in the configuration file. After this step, conventional hardware design tools can be used for simulation, synthesis, and post-synthesis analysis. For the software toolflow, an efficient compiler is developed [135], which supports both C and

Table 2.1: Configurations of the target architecture.

Property	Options
Bypassing	{implicit, explicit}
Datapath width	32b
Pipeline stages	4
Number of PEs	{8, 16, 32, 64, 128}
IMEM	56b × 2k
PE DMEM	32b × 1k
CP DMEM	32b × 1k

OpenCL. This compiler takes the same architecture configuration into account and produces the proper binary. Furthermore, a cycle-accurate simulator is generated based on the configuration file.

2.3 Experimental Setup

This section describes the experimental setup used to quantify the efficiency of the proposed architecture in general, and the effectiveness of an explicit datapath in th context of a wide-SIMD architecture. Section 2.3.1 describes the configurations of the target architecture as well as a reference RISC architecture. Section 2.3.2 presents the benchmarks used for the evaluation.

2.3.1 Architecture Configurations

To evaluate the effectiveness of the proposed design, both the explicitly and implicitly bypassed versions of the SIMD architecture proposed in section 2.2 are implemented in HDL. The configurations used in the experiments are shown in table 2.1. Since the proposed architecture is configurable, the HDL code of each configuration is automatically generated from an architecture template. For a complete analysis, the proposed SIMD architecture is also compared to a reference RISC architecture. The configuration of the reference RISC processor is given in table 2.2.

To exclude any interference a memory hierarchy could introduce in the measurements due to data misses, the evaluated designs assume only one level of memory. This memory can be accessed within a single cycle. The sizes of the data memories of the RISC and SIMD are chosen such that all data of the benchmarks can be contained.

Table 2.2: Configuration of the reference RISC processor.

Property	Setting
Datapath width	32b
Pipeline stages	4
IMEM	24b × 4k
DMEM	32b × 4k

Table 2.3: Energy dissipation of different memory accesses.

Mem. Config	32b × 1k	24b × 4k	32b × 4k	56b × 2k
PJ/Access	2.02	2.49	2.92	3.37

The core part of each configuration, that is the whole system with the exception of the memories, is synthesized for 1.1V, 25 °C, typical case, with a 40 nm commercial low-power complementary metal oxide semiconductor (CMOS) digital standard cell library. The target frequency is set to 100 MHz. Energy dissipation is estimated using the physical information in the technology library and circuit toggle rate generated by post-synthesis simulation on the gate-level netlist. The energy dissipation of the memory part is estimated with CACTI [149]. The CACTI tool provides an average access energy for a given memory configuration. The estimated access energy of the corresponding memory configuration is given in table 2.3. The number of accesses to each memory is extracted from simulation.

2.3.2 Benchmarks

To have a comprehensive evaluation of the proposed design across various types of applications, a total of eleven representative kernels are chosen, which are divided into four categories based on their communication and memory access patterns (table 2.4). The four kernel categories are:

1. *point-to-point*
   Binarisation is a typical example of this type of kernel. To calculate an output at $D_{out}(x, y)$, a PE only needs to read the input data, $D_{in}(x, y)$, at the same location. Since this type of kernels does not need to access data in the other memory banks, no communication to neighbouring PEs is required.

2. *neighbourhood-to-point*
   To calculate an output at $D_{out}(x, y)$, a PE only needs to read the current

Table 2.4: Benchmark categories and their kernels.

point-to-point	**global-to-point**
Binarisation	Find Max. Element in a Vector
Colour Conversion	Sum of Vector Elements
	Vector-Vector Addition
neighbourhood-to-point	**global-to-global**
Convolution 3×3	Matrix Rotation
Erosion 3×3	Matrix Mirroring
5-tap FIR	Matrix Transpose

input data, $D_{in}(x, y)$, and its surrounding data, $D_{in}(x \pm m, y \pm n)$, where m and n are small (≤ 7). These data are either in the PE's own memory bank, or in its nearby memory banks, so only local communication is required. A 3×3 *low-pass filter* is an example of this type of kernel.

3. *global-to-point*
   To calculate an output at $D_{out}(x, y)$, a PE needs to read the data far away from its corresponding input data, i.e., global communication. This global communication is either in the X direction, i.e., $D_{in}(x \pm p, y)$, or in the Y direction, $D_{in}(x, y \pm q)$, where p and q are large (> 7). One paradigm that falls into this category is *reduction*, in which all the elements of a vector are combined into a single element by a certain operator such as add, min/max, logic and/or.

4. *global-to-global*
   To calculate an output at $D_{out}(x, y)$, a PE needs to read the data far away from its corresponding input data, i.e., global communication. This global communication can be in both the X and Y directions, i.e., $D_{in}(x \pm p, y \pm q)$, where p and q are large (> 7). For this type of kernels, every input data element is still mapped to an output data element, i.e., no reduction operator is involved. These kernels are often mapped in such a way that one dimension is handled when reading the required input data, and the other dimension is handled when writing the output. *Matrix rotation by* 90 *degrees* is an example of this type of kernel.

Evaluation based on such patterns is interesting in context of the proposed wide SIMD with neighbourhood communication network. This is because kernels with only local communication can be efficiently mapped onto such a design, while kernels with global access will spend a significant amount of cycles on

data transfers between PEs that are far apart. To reduce the overall cost of long-distance communication, two custom reduction algorithms are employed: *pipelined reduction* and *diagonal access reduction*. These algorithms do not rely on complex communication networks or any dedicated hardware. Chapter 3 further introduces these algorithms in detail. The key idea of both approaches is to utilize inter-vector parallelism instead of intra-vector parallelism, which can be applied to both *global-to-point* and *global-to-global* kernels.

Besides kernel-level evaluation, we also map the fast focus on structures (FFOS) application [58] to profile the proposed architecture. FFOS is the complete vision processing pipeline of an industrial application; organic light-emitting diode (OLED) screen printing. Its purpose is to find the centre of OLED cells at high speed in the manufacturing process. It consists of the following four parts:

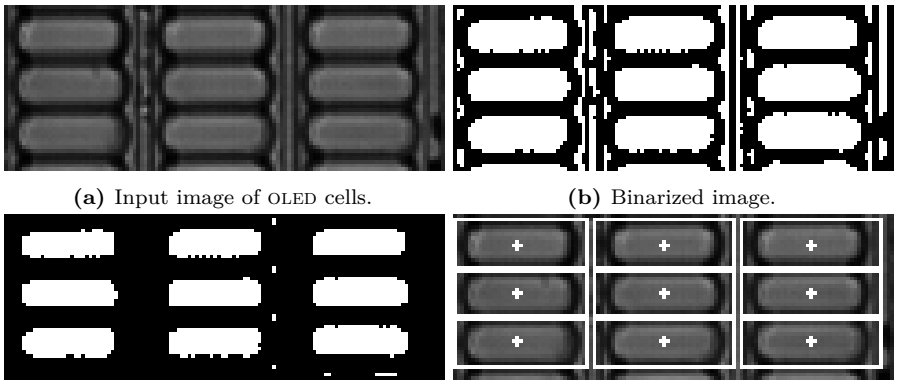
1. *Otsu*
   With the input image shown in figure 2.6a, the optimal threshold for binarisation is determined by means of Otsu's method [112]. Otsu's method exhaustively searches for a class that minimizes the intra-class variance. In order to achieve this, partial histograms per column are first calculated in the PE-array, which are then merged into a combined histogram. The optimum threshold is calculate based on this histogram. In order to achieve this 256 divisions are required, which are performed in parallel on the PE-array.

2. *Binarisation*
   Once the optimal threshold is determined, the input image is binarized to value 0 or 1. The result of this process can be seen in figure 2.6b.

3. *Erosion*
   In order to remove noise and small objects from the binarized image, an erosion kernel is applied to the binarized image. The eroded output image is shown in figure 2.6c.

4. *Row and column projection*
   Finally, by projection, i.e., summation, of both the rows and columns and determining the local peaks in the resulting vectors, the boundaries of the OLED cells can be determined. In figure 2.6d, bounding boxes are drawn around the detected cells.

As the number of cores scales up according to the PE parameters in table 2.1 it is important to decide how to scale the problem size accordingly. One option is to keep the problem size fixed while the number of parallel cores scales. This



(a) Input image of OLED cells. **(b)** Binarized image.

(c) Eroded image. **(d)** Bounding boxes.

Figure 2.6: FFOS images at different stages of the algorithms. Centres of the bounding boxes are obtained through row/column projection and subsequently determining the centres of the projected regions.

methodology is in line with Amdahl's law [8]. Amdahl's law predicts a maximum speedup as given by equation (2.1), where s is the sequential fraction of a program, p is the parallel fraction and N is the number of cores in the system. The implication of this law is that the achieved speedup rapidly diminishes if s is not very small. Therefore, if applied to the proposed SIMD, the expected speedup for 128 cores is limited by s.

$$\text{Speedup} = \frac{1}{s + {}^{p}/_{N}} \tag{2.1}$$

Not only is the speedup predicted by Amdah's law limited, it is also unrealistic for practical purposes as reasoned by Gustafson [46]. Gustafson argues that for realistic applications of multi-core systems, it is unlikely that the problem size is kept constant. Instead, a higher number of cores is typically used to solve bigger problems. For example in video processing a higher resolution can be used, or a weather prediction system can be applied to a larger area. Assuming the problem size scales with the number of cores leads to an alternative to Amdahl's law know as Gustafson's *scaled speedup*, which is given in equation (2.2).

$$\text{Speedup}_{\text{scaled}} = N + (1 - N) \cdot s, \tag{2.2}$$

where N represents the number of processing cores, and s is the serial fraction of the application. Because Gustafson's scaled speedup is arguably more appropriate

for real-life applications than Amdahl's law, in this chapter we choose to scale the problem size with the number of cores. In particular the kernels operate on an $N \times N$ matrix, where N is the number of PEs. Note that Gustafson postulates that the amount of work scales linearly with the number of processors, yet in this work the input is scaled quadratically with the number of PEs. This is done because for image and matrix oriented benchmarks this is a more natural choice. When both the width and height are scaled with N, the mapping of the problem to the processor can remain unchanged. For example if the row of an input image matches the number of PEs, it will match across all scaled versions. Otherwise some form of wraparound would be required, which would introduce a variable overhead in some mappings, but not in others. This would lead to unwanted noise in the speedup measurements, and skew any speedup results.

As an exception to the quadratic scaling, the input to FFOS is chosen to scale linearly. Since FFOS detects the centres of OLEDs in a production process, it is only interesting to increase the number of cells which are detected in the width of the input. The cells are moved underneath the camera so it is not useful to detect cells at a large number of rows simultaneously, yet detecting the centres of all cells in a row is the real goal of the application. Therefore it operates on input images of size $1024 \times N_{\text{PE}}$, where N_{PE} is the number of PEs. In this way the FFOS application is modelled in the most realistic way, and remains true to Gustafson's original approximation that the problem size increases linearly with the number of cores.

2.4 Results and Analysis

In section 2.4.1, the proposed architecture is first compared to a reference RISC processor in terms of area, performance, and energy dissipation. Performance and energy dissipation figures are obtained using the four types of kernels described in section 2.3.2. The purpose of this comparison is to demonstrate the scalability and energy efficiency of the proposed architecture. To examine the energy and performance impact of explicit bypassing in SIMDs, the proposed explicitly bypassed architecture is compared to its implicitly bypassed counterpart in section 2.4.2. The energy and performance analysis are done for all kernel categories as well as a realistic application, FFOS. By varying the number of PEs, the effects of explicit bypassing in an SIMD architecture are discussed in detail.

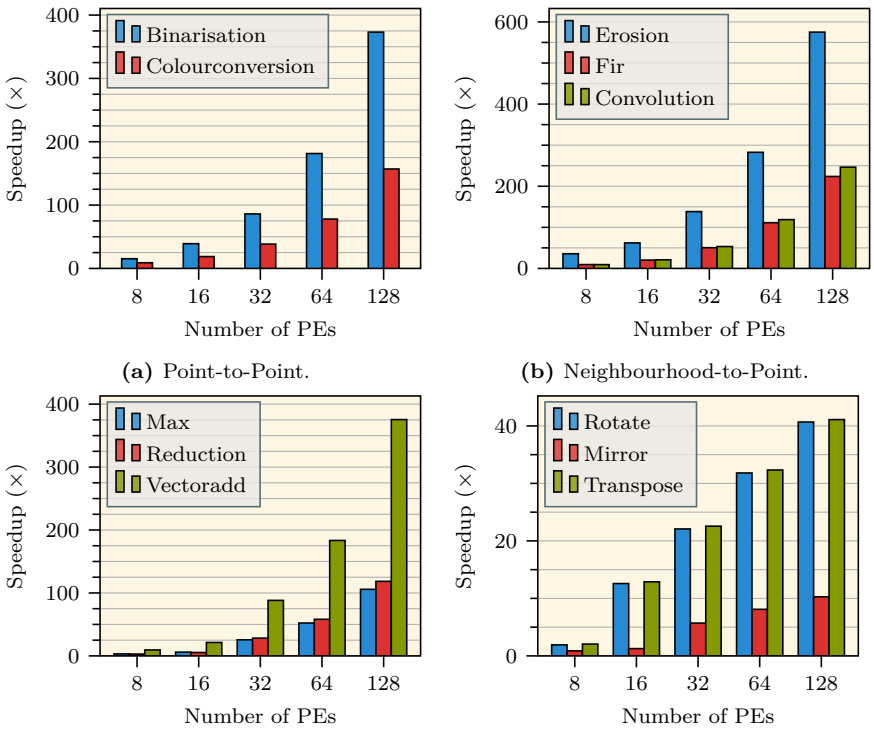
2.4.1 SIMD versus RISC

In this section the performance, area, and energy dissipation of the proposed explicit SIMD architecture is compared with that of a reference RISC architecture, which is described in section 2.3.1.

In terms of runtime the proposed SIMD has a significant speedup over the RISC in each kernel category as is shown in figure 2.7. For the *point-to-point* kernels as can be seen in figure 2.7a, the relation between the number of PEs and the speedup is almost linear. This is expected as the point-to-point kernels have no data dependencies between different PEs.

Figure 2.7b shows the speedup for the *neighbourhood-to-point* kernels. Interestingly, the speedup of the neighbourhood-to-point kernels is greater than that of the point-to-point kernels. This is because instead of only exploiting more DLP by adding more PEs also data reuse is exploited in a more efficient manner. If a neighbourhood is to be processed on the RISC, the processor will load every pixel inside the neighbourhood window and calculate a result. When the window shifts a couple of pixels can be reused and do not need to be reloaded, but also some pixels will be lost that are needed for a future computation. These pixels will have to be reloaded by the RISC once an overlapping window is processed. On the SIMD the image is loaded row by row and all windows in a line are processed in parallel. Since each PE has its own RF it is possible to keep a couple of complete rows in the RF at the same time. Therefore, once a pixel is loaded, it is used in all relevant neighbourhood calculations it belongs to and does not need to be loaded again. This saves extra operations and memory accesses which will also be visible later in the energy comparison.

For the *global-to-point* category the speedup is typically less than point-to-point and neighbourhood-to-point kernels as can be seen in figure 2.7c. Especially the *max* and *reduction* kernels have a lower speedup. This is explained by the fact that when the number of PEs increases, the data from these kernels need to travel further to reach their destination. Both the *max* and *reduction* kernels operate on data which is spread out across the PEs. More PEs means a longer path trough the neighbourhood network which somewhat counteracts the gain of exploiting more parallelism. The *vectoradd* kernel is the exception in this category. This is because the *max* and *reduction* kernels have data movement across the PE-array, i.e., reduction of the elements in a row, while the *vectoradd* kernel only has data movement within the PEs, i.e., reduction of the elements in a column. While the *max* and *reduction* kernels combine an element from every PE in the array,



(a) Point-to-Point.

(b) Neighbourhood-to-Point.

(c) Global-to-Point.

(d) Global-to-Global.

Figure 2.7: Speedup of the proposed SIMD architecture with respect to a baseline RISC machine. The communication dominated global-to-global kernels scale very poorly compared to the other kernel types.

Table 2.5: Relative core area of explicitly bypassed SIMD compared to RISC.

risc	Explicit Bypassed simd				
	8	16	32	64	128
1.00×	8.16×	15.2×	29.4×	57.8×	115×

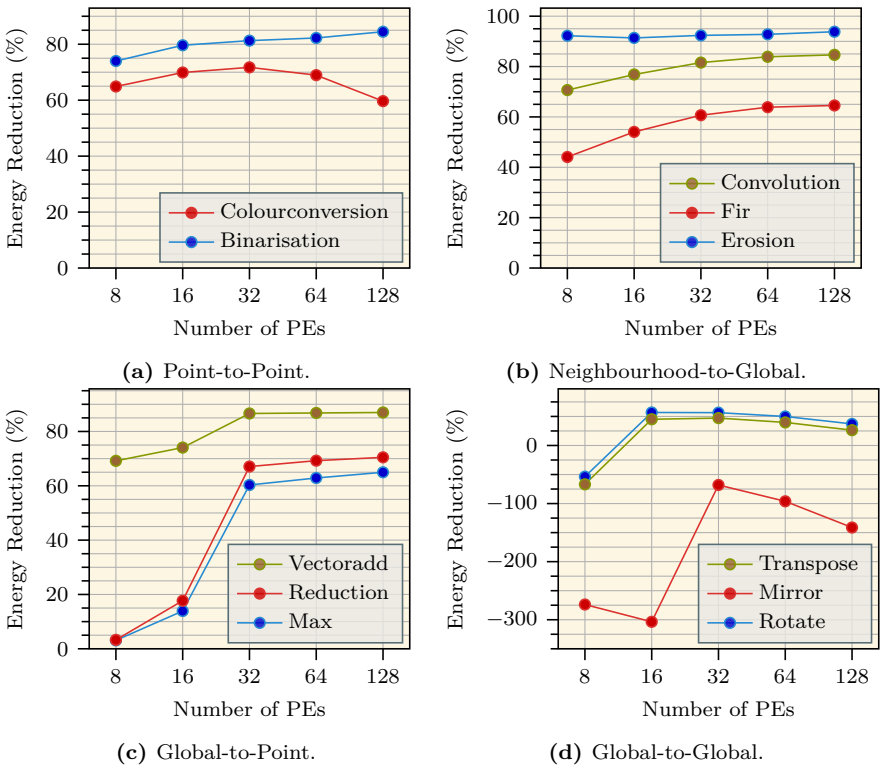
the *vectoradd* kernel only combines elements which are within the DMEM of a PE. Therefore, when the array becomes larger, there is no communication penalty such as the ones for the *max* and *reduction* kernels. As mentioned in the previous sections, to reduce the overall cost of long-distance communication, two reduction algorithms are introduced that exploit inter-vector parallelism instead of intra-vector parallelism, which will be further detailed in chapter 3. These approaches can also be applied to the *global-to-global* kernels.

Finally the *global-to-global* kernels show the least amount of speedup by adding more PEs, as is shown in figure 2.7d. For these kernels data elements need to move both between PEs and inside the data memories of the PEs, i.e., row- and column-wise. The communication patterns that arise from this are not always regular, making the neighbourhood network the main bottleneck when more PEs are added. Because of the more irregular patterns the global-to-global kernels pay an even higher penalty.

The core area of different instantiations of the explicitly bypassed SIMD, as well as that of the reference RISC processor, are shown in table 2.5. The area of the 8-PE SIMD is slightly larger than eight times that of the RISC processor. This is because an 8-PE SIMD consists of a vector array of eight PEs and a CP. The area of a PE itself is smaller than its RISC counterpart as the IF and part of the ID logic are shared among all PEs. When the number of PEs increases, the CP area is amortized over more PEs. Table 2.5 shows that the proposed SIMD architecture scales well in area.

In this analysis the energy of both the core and memory are considered. The reduction of the overall energy dissipation of the proposed SIMD architecture compared to the reference RISC is shown in figure 2.8.

Figure 2.8a shows the results of the *point-to-point* kernels. It can be seen that the reduction in energy dissipation of the binarisation kernel keeps on increasing when more PEs are used. For the colour conversion kernel, however, increasing the number of PEs beyond 32 starts to degrade the efficiency. It seems that for the colour conversion kernel after a certain number of PEs the energy overhead



(a) Point-to-Point.

(b) Neighbourhood-to-Global.

(c) Global-to-Point.

(d) Global-to-Global.

Figure 2.8: Energy dissipation compared to RISC. Most kernel types demonstrate high energy reductions with the notable exception of global-to-global kernels where the energy efficiency is lower than that of the baseline.

of additional hardware, such as neighbourhood communication network and predication logic, is not compensated sufficiently by the speedup it provides.

For the *neighbourhood-to-point* kernels, it can be seen in figure 2.8b that increasing the number of PEs always leads to an increased energy efficiency. The reason for this is twofold: on one hand the speedup of these kernels is slightly higher than for the colour conversion kernel. On the other hand, because of better exploitation of the locality of the data, the number of external memory accesses of the neighbourhood-to-point kernels decreases more than for the colour conversion kernel when the number of PEs increases. Accessing the DMEM is expensive in terms of energy, so reducing the accesses to this memory has a profound effect on the overall energy dissipation.

The *global-to-point* kernels seem to benefit especially when going from a relative low number of PEs to a higher amount. In particular the kernels that gather elements from all different PEs exhibit this effect. In figure 2.8c, the biggest increase in energy efficiency is observed when increasing the number of PEs from 16 to 32. For lower numbers of PEs, there is only a small reduction of the energy dissipation. The overhead of the control of the array is hardly compensated by the exploited DLP at this point. When the number of PEs increases, more DLP can be exploited, while the control overhead remains similar. This leads to an increased energy efficiency. When more and more PEs are added, the neighbourhood network starts to become a bottleneck. The positive effect of exploiting DLP is partly counteracted by the longer communication distances across the neighbourhood network.

Finally the *global-to-global* kernels show the least reduction in energy dissipation, which is shown in figure 2.8d. However, this is to be expected as their overall speedup as shown in figure 2.7d is also much less than the other kernel categories. More importantly, unlike the RISC processor, which can directly access its complete memory space, a PE within an SIMD processor requires extra operations to access data in other memory banks. This communication overhead significantly reduces the benefit of the increased exploitation of DLP. The mirror kernel is an extreme in these kernels, the SIMD actually performs worse than its RISC counterpart. These kernels exhibit a similar behaviour as the *max* and *reduction* kernels of the global-to-point category. When going up from a small number of PEs, first the efficiency is low. Then it increases due to the exploitation of DLP. And in the end this is counteracted by the increased communication distances across the PE array.

Overall, the examined kernels show a significant reduction in energy dissipation compared to the reference RISC machine. In all of these kernels, a significant speedup in performance is achieved in the proposed SIMD architecture due to efficient DLP exploitation. For a design that focuses on low energy, techniques like dynamic voltage frequency scaling (DVFS) can be applied to further improve the energy efficiency, while still meeting the same performance requirement as the RISC.

2.4.2 Explicitly versus Implicitly Bypassed

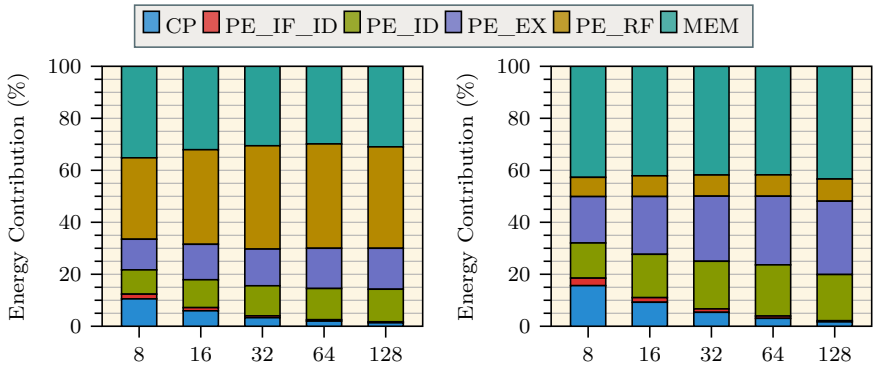
The goal of this section is to analyse the effectiveness of explicit bypassing over implicit bypassing in SIMDs. First energy breakdowns are presented which provide insight into where energy is being dissipated in both the explicit and implicit SIMD. Furthermore five aspects are discussed and analysed per kernel category, namely number of RF-accesses, RF energy dissipation, overall energy dissipation, performance, and area.

Energy Breakdowns

This section presents the energy breakdowns for one selected kernel out of each category in the benchmark. The breakdowns are given for each tested number of PEs and provide a comparison between the implicit and explicit bypassed SIMDs.

Each breakdown features six parts, the MEM accounts for the energy dissipated in both the IMEM and all the DMEMs. PE_RF represents the energy dissipated in the RFs of all the PEs. Similarly PE_EX and PE_ID represent the energy dissipated in the execution stage and local decode stage of all the PEs respectively. The PE_IF_ID category includes both the energy dissipated in the IF for the PEs, and the shared part of the instruction decode. Finally the part labelled CP represents the energy dissipated by the CP.

The energy breakdown for binarisation from the *point-to-point* category is given in figure 2.9. As expected the shared parts, such as the IF, global decode and CP become less important when the number of PEs increases. In figure 2.9a it can be seen that for the implicitly bypassed SIMD the RF starts to play a bigger role in the overall energy dissipation as the number of PEs increases. In the explicit SIMD the energy dissipation in the RF is reduced to such an extent that it is no longer the dominating part in the overall energy dissipation as can be seen in figure 2.9b.

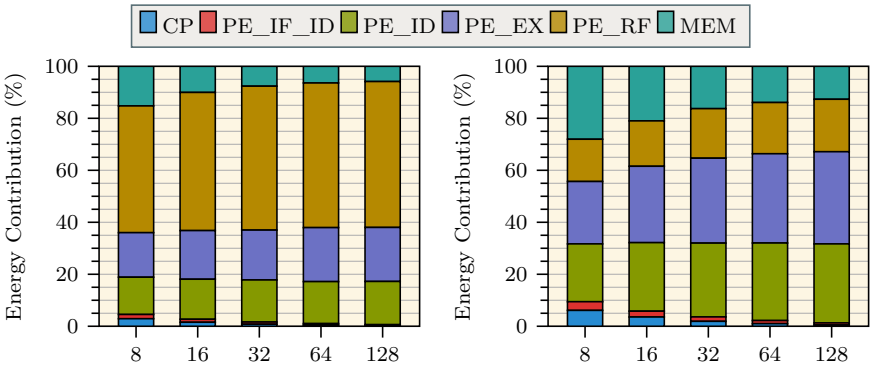


(a) Implicitly Bypassed SIMD. **(b)** Explicitly Bypassed SIMD.

Figure 2.9: Energy Breakdown for Binarisation. The energy consumed by the register file is heavily reduced by explicit bypassing.

For the *Convolution* kernel from the *neighbourhood-to-point* category, the energy breakdown is shown in figure 2.10. From the figure it is clear that the RF dominates the energy dissipation even more, especially in the implicitly bypassed SIMD. The *neighbourhood-to-point* kernels generally store data elements from the neighbourhood in the RF, and update it as the neighbourhood window slides over the input. This explains why the RF in this case is used more, and thus dissipates relatively more energy than the *point-to-point* kernels. The positive effect of explicit bypassing on the energy dissipation of the RF can be clearly seen in figure 2.10b. The RF still accounts for a larger portion of the total energy dissipation as the number of PEs increases, but the explicit datapath techniques significantly reduce the energy dissipation relative to the implicitly bypassed SIMD.

The breakdown of the *Vector-Vector Addition* kernel from the global-to-point category is given in figure 2.11. Comparable with the Convolution kernel, the *Vector-Vector Addition* kernel uses the RF heavily to exploit locality. This is why the RF again accounts for such a large amount of the total energy dissipation in the implicitly bypassed SIMD, as can be seen in figure 2.11a. However, the vast majority of variables in the *Vector-Vector Addition* kernel are short-lived, since each loaded element is just added to a sum without any other computations. This enables explicit bypassing to achieve high savings in energy dissipation, as a large number of accesses to the RF can be avoided. Given that the computation



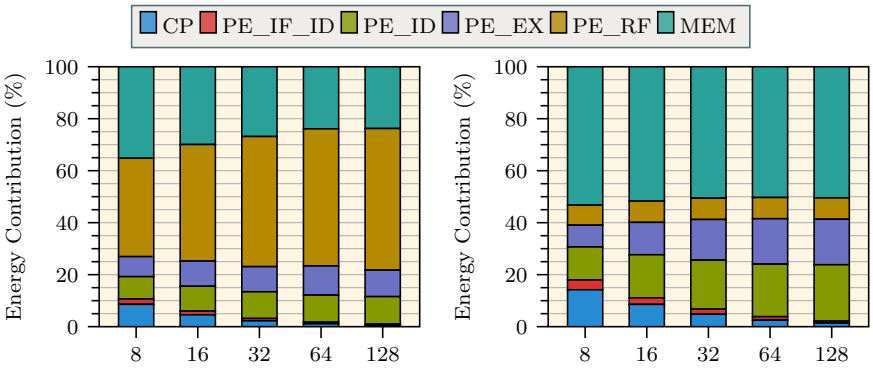
(a) Implicitly Bypassed SIMD. **(b)** Explicitly Bypassed SIMD.

Figure 2.10: Energy Breakdown for Convolution. The relative contribution of the register file in the implicit datapath is even larger than for the binarisation kernel.

of the *Vector-Vector Addition* is so simple, and the RF is almost not accessed due to explicit bypassing, the memory accesses dominate the energy dissipation of the explicit SIMD in figure 2.11b.

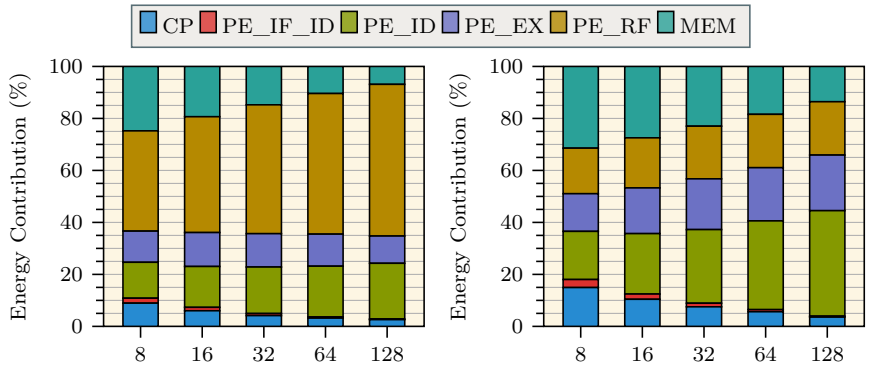
Finally, an energy breakdown is provided for *Matrix Transpose* from the global-to-global category. In this kernel data predominantly moves between the PEs. Since the PEs communicate by accessing each other's operands, every communication will result in a read and write of the RF in the implicitly bypassed SIMD. This is why also here the RF plays such a dominant role in the implicitly SIMD, as shown in figure 2.12a. Furthermore, when the number of PEs increases, so does the average communication distance. Therefore there are relatively more accesses to the RF for larger number of PEs, making the RF even more important for an increasing number of PEs. Explicit bypassing avoids a large amount of the RF accesses during long distance communication. After all, data is passed from neighbour to neighbour and never needs to be committed into the RF. This is why also here explicit bypassing is so effective at reducing the contribution of the RF to the total energy dissipation, as is visible in figure 2.12b.

Overall it can be seen that the RF plays an increasingly more dominant role in the implicitly bypassed SIMD when the number of PEs increases. Yet the explicit bypassing techniques significantly reduce the contribution of the energy dissipation of the RF. In the following sections the implicitly bypassed and



(a) Implicitly Bypassed SIMD. **(b)** Explicitly Bypassed SIMD.

Figure 2.11: Energy Breakdown for Vector-Vector Addition. The reduction of the contribution of the register file is highly similar to the binarisation kernel.



(a) Implicitly Bypassed SIMD. **(b)** Explicitly Bypassed SIMD.

Figure 2.12: Energy Breakdown for Matrix Transpose. The contribution of the register file grows as the vector size increases, and the communication distances grow.

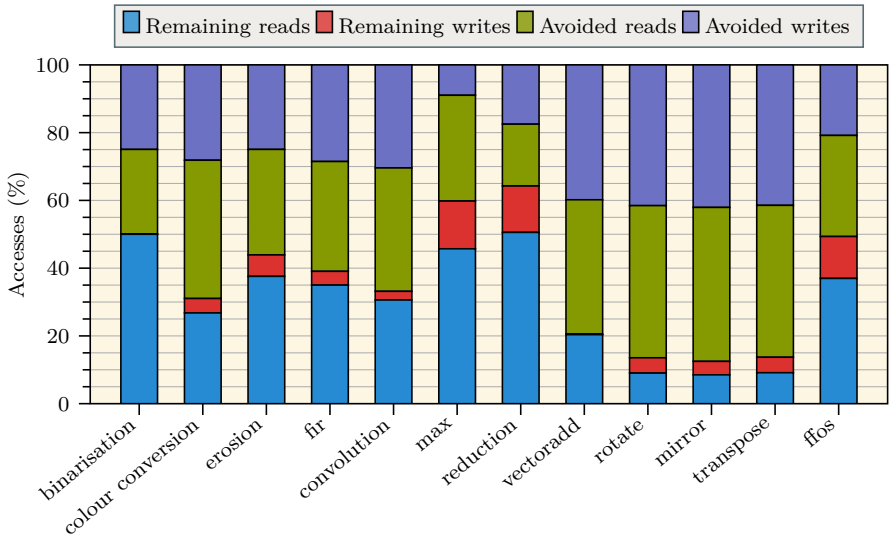


Figure 2.13: Breakdown of the RF accesses per benchmark. In particular the global-to-global kernels benefit significantly as long communication no longer needs to access the RF.

explicitly bypassed SIMD will be compared for each of the kernel categories with respect to the absolute number and type of RF accesses, energy dissipation in the RF, and the overall energy dissipation.

Point-to-Point

The main cause of reduction in energy dissipation in an explicitly bypassed SIMD processor is the reduction of traffic to the register file. Figure 2.13 shows how many RF accesses (both RF reads and writes) are avoided compared to the implicitly bypassed SIMD.

The point-to-point kernels only read and write to the private data-memory of a PE. Each data element is updated based on only its current value. Therefore, most variables are short-lived and can be bypassed. This is shown in terms of avoided accesses (figure 2.13). For the binarisation kernel, the number of remaining RF writes is even reduced to zero. Each pixel is loaded, compared to a threshold and written back to the main memory. The lifespan of the pixel is

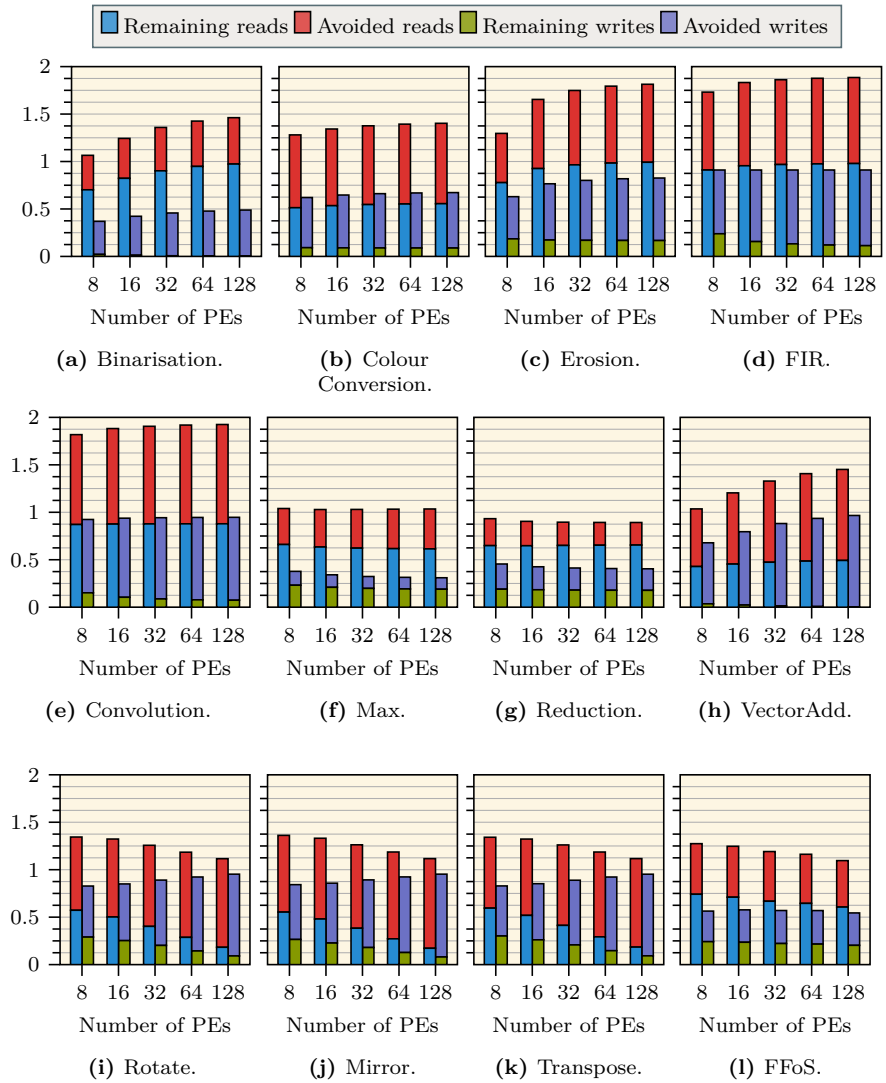


Figure 2.14: RF reads and writes per cycle. The neighbourhood-to-point kernels have a relatively high read activity, while the global-to-point kernels are on the other side of the spectrum.

short enough to avoid involving the RF. The RF is only used to hold memory addresses and the threshold for binarisation.

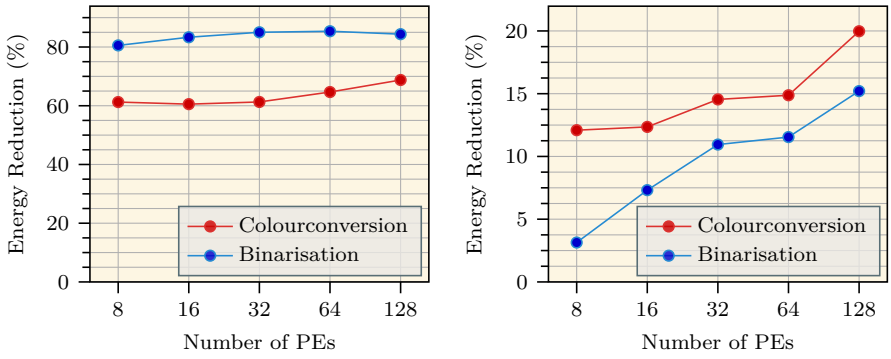
If a large kernel has only one RF access, avoiding that access hardly brings any reduction of energy dissipation. Therefore it makes sense to analyse the effectiveness of explicit bypassing by looking at the average number of read/write accesses per cycle in figure 2.14, rather than just the normalised accesses in figure 2.13. Figure 2.14a and figure 2.14b show the absolute number of reads and writes per cycle for the point-to-point kernels. The **red**/**violet** bars indicate the extra accesses required by the implicitly bypassed SIMD over the explicitly bypassed SIMD.

Figure 2.14a and 2.14b show that for the colour conversion kernel, relatively more reads are avoided than for the binarisation kernel. However, binarisation almost completely avoids all writes. Since RF writes consume more energy than RF reads, this explains why binarisation saves more RF energy as shown in figure 2.15a. For the overall energy dissipation, i.e., including the core, RF and memories, the colour conversion kernel has a higher reduction as shown. This is because the colour conversion has both a larger number of reads, and a larger number of writes per cycle to start with (figure 2.14b), which means the percentage of RF energy dissipation within the complete processor is higher. Although the reduction of energy dissipation in the RF is less than for binarisation, the total energy reduction in the colour conversion kernel is still higher.

Explicit bypassing results in significant reduction in energy dissipation for the point-to-point kernels. The reduction of overall energy usage increases when the number of PEs increases. This is because, as previously shown in section 2.4.2, the datapath, including the RF, of an SIMD processor plays an increasingly more important role in the overall energy dissipation when more PEs are added. This again shows that reducing the RF energy dissipation is particularly effective in (wide) SIMDs.

Neighbourhood-to-Point

For the neighbourhood-to-point kernels, behaviour similar to that of the point-to-point kernels is observed. Roughly around 60 to 70 % of the original RF accesses are avoided when explicit bypassing is applied in the erosion, FIR, and convolution kernels (figure 2.13). Out of these, the largest reduction in accesses is observed for the erosion kernel.

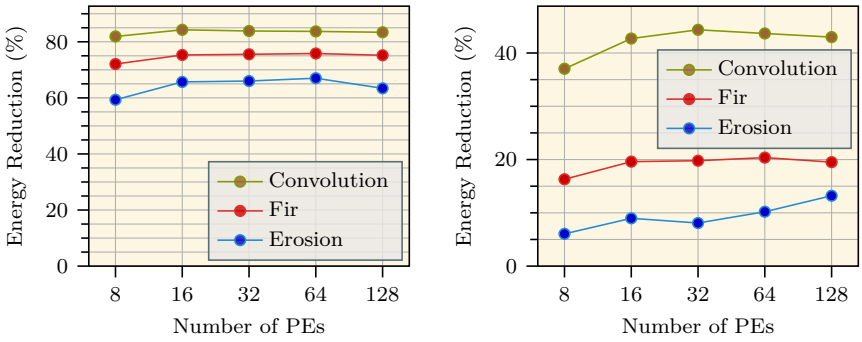


(a) RF energy usage reduction. **(b)** Overall energy usage reduction.

Figure 2.15: Energy usage reductions by explicit bypassing: point-to-point kernels. The gains on the RF are higher for binarisation, but the colour conversion kernel still profits most on the overall application.

This is also reflected in the number of accesses per cycle, as shown in figure 2.14c, 2.14d, and 2.14e. Among these kernels, the number of writes per cycle of the convolution kernel decreases the most. This is also the reason the convolution kernel has the largest reduction of energy dissipation, both in the RF and overall.

Compared to the point-to-point category, it can be seen in figure 2.15b and figure 2.16b that in the neighbourhood-to-point category the overall energy dissipation reduces the most. This is due to the inherent nature of the neighbourhood-to-point category. Since the kernels in this category gather surrounding pixels and merge them into a single value, a lot of short-lived variables exist. Pixels are moved around the neighbourhood and absorbed quickly. Moving the pixels around typically requires a large number of RF accesses in the implicitly bypassed datapath. In the explicit datapath, these short-lived variables provide an excellent opportunity to reduce RF accesses, incurring a large reduction in the overall energy usage. This can also be observed by comparing the initial number of accesses per cycle of the neighbourhood-to-pixel category in figure 2.14c, 2.14d, and 2.14e to the corresponding figures of the other kernel categories. No other category has such a large amount of accesses per cycle in the implicit datapath and reduces the number of accesses by this much.

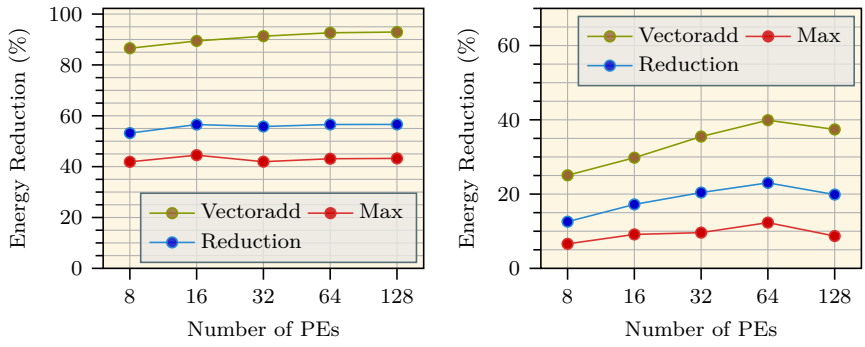


(a) RF energy usage reduction. **(b)** Overall energy usage reduction.

Figure 2.16: Energy usage reductions by explicit bypassing: neighbourhood-to-point kernels. Despite the very similar read/write intensities shown in figure 2.14, the overall energy benefits are spread out much more than the gain on the RF would suggest.

Global-to-Point

From the register file access numbers of the global-to-point kernels, i.e., max, reduction, and vectoradd, in figure 2.13, it can be seen that the max and reduction kernel both reduce the number of accesses by a significant amount. The vectoradd kernel is an outlier however, and avoids almost all accesses. This is because the max and reduction kernels combine data elements that are spread out across the PE-array. Therefore, they cause a large amount of communication and control overhead, in order to coordinate the data transfers. The vectoradd kernel, however, only combines data elements that are already located in the same PE data memory. Because all data elements are already located in the private data memory of a PE, the values only need to be loaded and added to a sum variable located in the RF. In the implicit datapath, each load induces a write to the RF, and summing the loaded value causes two reads. Since loading and adding a pixel can be done in just a couple of instructions, these accesses to the RF can be almost completely avoided. This is why in figure 2.13 vectoradd reduces the RF accesses much more than the other two global-to-point kernels. It is therefore no surprise that the vectoradd kernel reduces the energy usage most of all kernels in the global-to-point category (figure 2.17). The energy dissipated in the RF is significant with implicit bypassing for the vectoradd kernel, but with explicit bypassing the energy used in the RF is reduced by more than 90 %.



(a) RF energy usage reduction. **(b)** Overall energy usage reduction.

Figure 2.17: Energy usage reductions by explicit bypassing: global-to-point kernels. There appears to be a global optimum at 64 PEs, after which the overhead of explicit bypassing is starting to dominate the added gains.

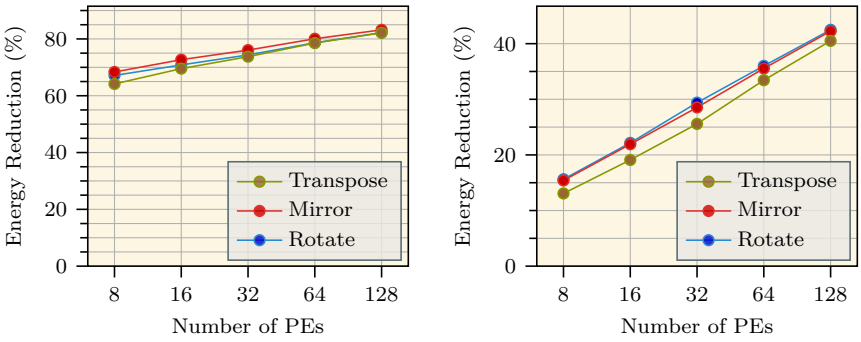
Global-to-Global

The global-to-global kernels require a large amount of long distance communication. In PE-to-PE communications, variables only pass through the PEs, hence they do not need to be stored in the RF. This is the reason that explicit bypassing avoids a significant amount of the RF accesses in the global-to-global kernels, as can be seen from figure 2.14i, 2.14j, and 2.14k. When the number of PEs increases, the percentage of PE-to-PE communications increases accordingly. This directly translates into an increasing energy efficiency for the global-to-global kernels, as is shown in figure 2.18.

FFoS

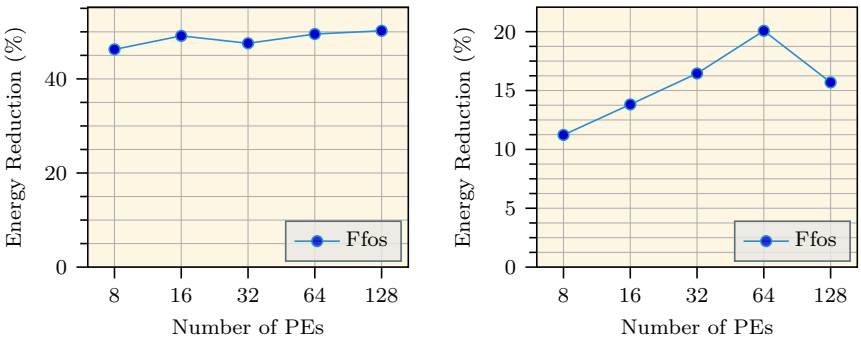
In this section the industrial FFOS application is benchmarked. The size of the input image is $1024 \times N_{\text{PE}}$, where N_{PE} is the number of PEs in a particular SIMD instantiation. Because the number of rows of the input image is fixed, the number of avoided RF reads and RF writes per cycle is hardly influenced when more PEs are added (figure 2.14l). As a result, the reduction of the energy dissipation in the RF is around 48 %, as is shown in figure 2.19a.

The FFOS application is particularly memory intensive. In order to clearly show the effects of explicit bypassing, which does not affect energy used in the data memories, figure 2.19b only shows the reduction in energy dissipation in the



(a) RF energy usage global. **(b)** Overall energy usage global.

Figure 2.18: Energy usage reductions by explicit bypassing: global-to-global kernels. The gains of these kernels grows linearly with increasing vector width. This happens because the communication distance in these kernels grows proportionally to the vector width as well, and without explicit bypassing communication traverses through the RF.



(a) RF energy usage global. **(b)** Overall energy usage application.

Figure 2.19: Energy usage reductions by explicit bypassing: FFOS. The overall energy gains follow the patterns of the global-to-point kernels (figure 2.17), with an optimum at 64 PEs.

Table 2.6: Cycle count of each kernel on 128-PE SIMDs.

Kernel	Explicit	Implicit
Binarisation	527	526
Colour Conversion	4383	4382
Erosion 3×3	747	747
5-tap FIR	1411	1411
Convolution 3×3	2452	2452
Find Max. Element in a Vector	1100	1099
Sum of Vector Elements	844	843
Vector-Vector Addition	268	268
Matrix Rotation	5671	5542
Matrix Mirror	11273	11015
Matrix Transpose	5606	5477
FFOS	32961	32961
Average performance loss	0.62 %	

core/logic part of the processor. It is interesting that the FFOS application shows an overall improved energy efficiency for the logic part when the number of PEs is increased, even though the number of avoided RF reads and writes per cycle is hardly influenced. This is due to the fact that the register file, percentage-wise, contributes a larger part to the total energy dissipation, because the instruction fetch and decode are amortized over more PEs. This makes the register file's contribution to the energy dissipation larger, so even though the reduction in the register file is nearly constant, overall the energy usage reduces as the number of PEs increases.

Performance

Table 2.6 shows the cycle count of each kernel on both the 128-PE SIMD with explicit bypassing and the 128-PE SIMD with implicitly bypassing. Because explicit datapath architectures maintain state in the pipeline, sometimes additional instructions are required to explicitly flush this state at the points where control flow merges. The results in table 2.6 shows however, the overhead of these additional cycles is almost negligible for the proposed architecture.

Area

The core area of the implicitly bypassed SIMDs is shown in table 2.7. Compared to table 2.5, it can be seen that the explicitly bypassed SIMDs occupy slightly

Table 2.7: Core area of implicitly bypassed SIMD compared to a single-core RISC.

risc	Implicit Bypassed simd				
	8	16	32	64	128
1.0×	8.31×	15.6×	30.3×	59.3×	117×

less area. This is because the explicitly bypassed SIMDs have slightly smaller physical register files and simpler bypassing logic.

2.5 Related Work

Reducing the energy dissipation of the register file has always been considered important in improving processor energy efficiency [57, 172]. About 15 % of the core energy within a typical single-issue RISC processor is dissipated by the register file, and an even higher percentage for processors that exploit more instruction-level or data-level parallelism [56, 163, 172]. Earlier work has shown that optimizing the bypassing network can reduce this large energy usage [45]. For example, in VLIWs it has been shown that storing short-lived values in pipeline registers can reduce energy usage while sustaining the compute performance [39]. Similarly, in a transport triggered architecture (TTA), which is considered to be a superset of the VLIW architecture [29], the reduction of energy dissipation in the RF induced by explicit bypassing has been shown to be as much as 80 %, leading to a reduction of the overall energy dissipation of 11 % [57]. A compiler was developed for this TTA [134]. It fully automates explicit bypassing and achieves the same amount of energy reduction. This proves the practical value of explicit bypassing. However, none of the related works provide a detailed head-to-head comparison in terms of energy efficiency between explicit and implicit bypassing in an SIMD setting.

Explicit bypassing is also used to improve performance by mitigating RF pressure on both size and number of read/write ports. The MOVE work [31] and the TCE work [146], both of which are TTAs, studied this thoroughly. Yan et al. introduce a similar concept, called virtual register, which exploits the short-lived variables and the data bypassing network to minimize the demand on real registers [179]. Instead of focusing on power dissipation, this work is mainly aimed at achieving higher performance without enlarging the RF physically. Compared to this work, the proposed architecture exploits the same principle, but with a more

compact instruction format, resulting in smaller instruction memory and less expensive memory access in terms of energy dissipation. Moreover, in the proposed architecture, data stays available longer for bypassing. This is because input latches are introduced to each functional unit (FU), such that an FU output is preserved till the same FU is used by another instruction. Since data is available longer for bypassing, more variables can be bypassed, reducing the traffic from/to the RF [57].

Wide SIMD architectures are widely used in embedded processors. The Xetal from NXP [2] is an SIMD processor with 320 PEs that is designed for smart camera data processing. The PEs in Xetal are connected by a neighbourhood network. However, Xetal has one large vector memory (called frame memory). It does not have any small memory (e.g., a register file) in between to exploit data locality. Because of this, energy dissipation of data accesses is high. He et al. address this issue in Xetal-Pro [56, 118]. By introducing an extra level of memory, as well as aggressive voltage scaling, a much more efficient architecture is obtained. This work shows that to achieve ultra-low power, improving the efficiency of data movement is of crucial importance in SIMDs, which motivated the introduction of explicit datapath techniques in this chapter. The IMAPCAR from NEC [84] is another example of a wide SIMD processor. The IMAPCAR has 128 PEs connected with a ring network. A key difference in IMAPCAR compared to Xetal is that it has independent address generation for each PE. While the memory is more complex in such a configuration, it also results in much better programmability. Since many applications, such as histogram and Hough transform, can benefit from independent address generation [59, 60], the proposed architecture supports independent address generation for each PE.

Woh et al. propose AnySP, a wide SIMD targeting wireless and multimedia applications [172]. The PE interconnect in AnySP is a reconfigurable RAM-based crossbar, which is more flexible compared to Xetal, IMAPCAR, and the work presented this chapter. The energy usage of the vector register file in AnySP is reduced by introducing an extra 4-entry small register file. AnySP also uses explicit bypassing. However, instead of using a small RF to increase the bypass opportunities and reduce the RF pressure, we achieve these goals by increasing the number of bypassing sources in the proposed architecture.

In another work of Woh et al., the evolution from SODA to Ardbeg is presented [173]. It is noted that the RF is the largest power consumer in SODA, accounting for 30 % of the total power. To mitigate this problem, Ardbeg introduces 2-issue long instruction word (LIW) support, allowing a restricted set of

operations to run in parallel. In order to facilitate LIW, the RF requires two read and two write ports, making the RF more complex, and therefore presumably more power hungry. Yet the performance gained by the 2-issue LIW results in an overall better energy-delay product. This technique is orthogonal to the explicit datapath approach evaluated here, and it would be interesting to investigate how much energy can be saved by combining the two techniques.

In our work, the proposed architecture is similar to the Xetal-Pro [56]. The main differences are that the proposed architecture uses per-PE register files, independent addressing, and a PE datapath with explicit bypassing. Compared to the PE micro architecture of Xetal-Pro, which supports limited operation types due to its simplicity [56], the PE micro architecture of this chapter is RISC-like and supports more operation types.

2.6 Conclusions

In this chapter a low-energy wide-SIMD architecture with explicit datapath is proposed. The proposed architecture is fully programmable and features a configurable number of processing elements and pipeline stages. Scalar operations and (wide) vector operations are issued in parallel to exploit DLP and ILP at the same time.

To show the effectiveness of the proposed architecture an instantiation of the explicitly bypassed architecture with 128 PEs is compared with a reference RISC architecture. The experimental results show that the SIMD processor reduces the energy dissipation by up to 94 % in the erosion kernel and by 48.3 % on average for the total of eleven tested kernels. The proposed SIMD processor also achieves an average of 206× speedup compared to the reference RISC even though it only has 128 PEs. This is because in the proposed SIMD architecture scalar operations and (wide) vector operations are issued in parallel to exploit DLP and ILP at the same time and enhanced exploitation of data locality.

To demonstrate the effectiveness of explicit bypassing in an SIMD environment multiple instantiations of the proposed architecture are implemented. Eleven representative kernels and one industrial application are mapped onto all these instantiations, as well as their implicitly bypassed counterparts. Detailed comparison and analysis are carried out. The experimental results show that compared to the implicit bypassing counterpart a considerable number of RF accesses are avoided by using explicit bypassing. In particular 64 % on average for 128 PEs.

For total energy dissipation, an average of 27.5 %, and maximum of 43.0 %, reduction is achieved.

Chapter 3

Reduction Operator for Wide-SIMDs Reconsidered

This chapter is based on the work published in "Reduction operator for wide-SIMDs reconsidered" [158].

In various application domains, including deep learning, *reduction* is a frequently encountered operation, where multiple input elements need to be combined into a single element by an associative operation, e.g., addition or multiplication. There are many applications that require reduction such as: partial histogram merging, matrix multiplication, min/max-finding, and accumulation over neural network feature maps. To efficiently support reduction operations on the wide-SIMD with minimal interconnect as described in chapter 2, two novel reduction algorithms are introduced which do not rely on complex communication networks or any dedicated hardware. The proposed approaches are compared with both dedicated hardware and other software solutions in terms of performance, area, and energy consumption. A practical case study demonstrates that the proposed software approach has much better generality, and no additional hardware cost. Compared to a dedicated hardware adder tree, the proposed software approach saves 6.8 % in logic area with a performance penalty of only 7.1 %, while supporting a variety of reduction operations as opposed to only accumulation.

3.1 Introduction

Reduction is a higher order function which combines a given list of input elements through the use of an associative operation, constructing a single return value. Examples of reduction are calculating the sum of the elements of a vector, finding the maximum or minimum element in a list, and logic operations such as `and` and `or` over a vector. Reduction is encountered so frequently that many programming languages such as C++, Python, OCaml, Perl and Ruby, have built-in support for it, although often under different names including `reduce`, `fold`, `aggregate`,

`compress` and `inject`. Amongst others, reduction is required for kernels such as Partial Histogram Merging, Sum of Absolute Differences, Row Projection, Min/Max-finding, Matrix Multiplication, and 2D Convolution layers in neural networks.

Because the operator used in reduction is associative, the different combine operations can be performed independently. Thus, reduction inherently possesses a large amount of data-level parallelism (DLP). This DLP can be exploited by the wide-single instruction multiple data (SIMD) architecture introduced in chapter 2. Given that reduction is such an important part of the target domains of this wide-SIMD, and plays such a dominant part in the evaluation of convolutional neural networks, it is imperative to support reduction in an efficient manner. This is particularly challenging given the minimal interconnect present on this SIMD, as described in section 2.2.2.

In this chapter two reduction algorithms optimized for wide-SIMDs with minimal interconnect are proposed. These algorithms do not rely on any additional hardware and require only local communication with short wires, making this approach extremely scalable. Furthermore this software approach is completely agnostic to the type of reduction operation. To demonstrate the effectiveness of the proposed algorithm, implementations on the wide-SIMD with limited connectivity as presented in chapter 2 are compared to both a straightforward mapping and a solution with dedicated hardware. A case study shows that dedicated hardware is only 7.1 % faster, while it consumes 6.8 % more chip area.

The remaining parts of this chapter are organised as follows. First the context of the problem at hand is discussed in section 3.2. Next a single straightforward, and the two proposed reduction algorithms, are presented in section 3.3. The novel reduction algorithms are analysed and compared with reference approaches in section 3.4, including the results of a case study. Finally, related work and conclusions are provided in section 3.5 section 3.6 respectively.

3.2 Context

This section briefly describes the target platform used to benchmark the novel reduction algorithms. Furthermore the data layout on this platform and a dedicated hardware approach are described.

3.2.1 Target Architecture

The experiments in this chapter are based on the wide SIMD with explicit datapath introduced in chapter 2. Of particular interest is the circular neighbourhood network described in section 2.2.2 (page 27), which is an extremely minimal type of interconnect. In this network, all processing elements (PEs) are connected in a circular fashion as shown in figure 2.4 (page 27). To communicate, a PE is able to access one of its neighbouring PE's operands.

Note that the control processor (CP) can be a part of the loop or not, depending on the configuration of the first and last PE. It is also possible to 'break' the loop and let the boundary PEs read a predefined value. This configuration can be changed at runtime.

All the wires of this neighbourhood network are local and there is no complex/global network control involved, which make this network highly scalable. This scalability comes at the price of degraded performance for long distance communication. The key concept is that when a PE needs to exchange data with a PE not directly adjacent to it, that data will have to pass through all PEs in between. Every hop in this chain takes one cycle, hence long distance communication is slow and inefficient. Therefore the challenge of this network is to map algorithms in such a way that communication is kept local as much as possible.

3.2.2 Data Layout

The goal of the reduction techniques is to combine the elements of a vector which is distributed over the data memories of the PE array. In particular we assume N_{vect} vectors of size V_{size} elements are stored in the N_{PE} data memories of the target SIMD. The N_{vect} reduced outputs have to end up in the CP. In terms of data layout in the PE array two cases can be distinguished:

case 1, $V_{\text{size}} \leq N_{\mathbf{pe}}$**:**
If the vector size is smaller or equal to the number of PEs, each vector has at most one element in the data memory (DMEM) of each PE. The vectors are assumed to be stored in rows, and in case $V_{\text{size}} < N_{\text{PE}}$ the last PEs in the array are assumed to hold no elements and can be left out of consideration. In figure 3.1 the position of 4 vectors in the DMEM of the target architecture is illustrated.

case 2, $V_{\text{size}} > N_{\mathbf{pe}}$**:**
If the vector has more elements than there are PEs, a wrap around is required.

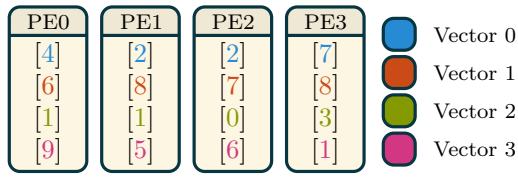
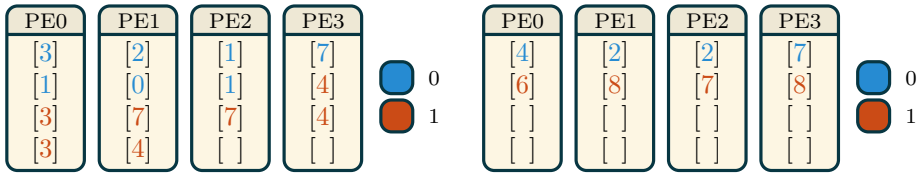


Figure 3.1: Case 1: $V_{\text{size}} \leq N_{\text{PE}}$.



(a) Initial situation. **(b)** After column reduction.

Figure 3.2: Conversion from case 2 to case 1 $V_{\text{size}} > N_{\text{PE}}$. N.B., here *addition* is arbitrarily chosen as the combine operator for illustration purposes. The choice of operation is completely free.

Therefore the DMEM of a PE will contain at least one element of the vector and possibly more. It is relatively easy to convert this case to case 1, by letting each PE locally reduce all elements associated to the same vector in its private DMEM. This leads to the same layout as in case 1 where each PE has one element per vector. The conversion from case 2 to case 1 is illustrated in figure 3.2.

The conversion from case 2 to case 1 is a simple procedure, since there is no communication required between PEs. Given that a PE contains a maximum of $\left\lceil \frac{V_{\text{size}}}{N_{\text{PE}}} \right\rceil$ elements of a single vector, converting case 2 to case 1 would take $\left\lceil \frac{V_{\text{size}}}{N_{\text{PE}}} \right\rceil$ loads, $\left\lceil \frac{V_{\text{size}}}{N_{\text{PE}}} \right\rceil - 1$ combine operations and 1 store operation. This gives a total of $2 \times \left\lceil \frac{V_{\text{size}}}{N_{\text{PE}}} \right\rceil$.

All the algorithms and techniques discussed hereafter assume a data layout as shown in figure 3.1. To compensate for the conversion from a layout such as in figure 3.2a, an additional $2 \times \left\lceil \frac{V_{\text{size}}}{N_{\text{PE}}} \right\rceil$ cycles should be added to all running times given in the remainder of this chapter.

3.2.3 Dedicated Reduction Hardware

To benchmark the novel reduction algorithms they are compared with dedicated reduction hardware. Although dedicated hardware is not as scalable as a software approach, and fixes the supported combine operation at design time, it has been used in the past in wide-SIMDs as will be discussed in section 3.5. Therefore it is important to compare the novel algorithms with such an approach.

Since dedicated hardware fixes the type of supported combine operations, a choice has to be made on what to support. Calculating the sum of the elements of a vector is one of the most common types of reduction, and can be found in many kernels. Therefore the focus is on this type of reduction and an adder tree is added to the target architecture as dedicated hardware.

The used adder tree is fully pipelined and can start a new computation every cycle. It is as wide as the PE array and contains $\lceil \log_2 N_{\text{PE}} \rceil$ stages. The adder tree inputs and output are memory mapped. The PEs can write input elements to the reserved address, and the sum of those elements can be accessed by the CP.

3.3 Software Approaches

This section contains three software approaches to map reduction to the target architecture. Straightforward reduction is an attempt to exploit the DLP within a single reduction operation, and is intended as a reference for the novel algorithms. The *pipelined reduction* and *diagonal access reduction* are the two novel algorithms that map reduction efficiently to the target architecture using no dedicated hardware extensions or complicated interconnect requirements.

3.3.1 Straightforward Reduction

In typical cases the DLP in a reduction operation is exploited by performing the operations in a tree-like fashion, i.e. all operations in one layer of a binary reduction tree are executed in parallel. The mapping of such a tree to the PE array is illustrated in figure 3.3. As can be seen in figure 3.3, directly mapping such a reduction tree onto the target architecture results in a mismatch with the neighbourhood network. Per cycle, data can only be transferred either one PE to the left or to the right. The red arrows in figure 3.3 require communication over more than one PE, resulting in additional cycles to perform the communication. Per layer of the tree, the branches become longer and the overhead increases.

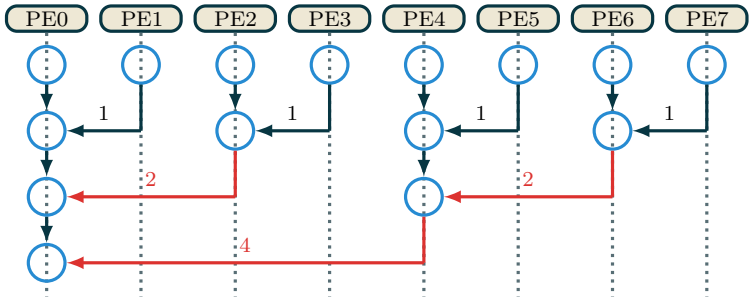


Figure 3.3: Reduction tree mapped to the PE array. The numbers indicate the number of required cycles to perform the communication. Red lines require more than one step and severely degrade the performance of the reduction tree.

The number of operations for layer i, consisting of one reduction operation plus communication operations is given in equation (3.1).

$$\text{OperationsPerLayer}\,(i) = 2^i, \text{with } i = 0, 1, \ldots \tag{3.1}$$

The number of layers in a reduction tree for vectors of size V_{size} is given in equation (3.2).

$$\text{layers}\,(V_{\text{size}}) = \lceil \log_2 V_{\text{size}} \rceil \tag{3.2}$$

Combining equation (3.1) equation (3.2), the number of required operations can be calculated, as is shown in equation (3.3).

$$\begin{aligned} &\text{Operations}(V_{\text{size}}) = \\ &= \sum_{i=0}^{layers(V_{\text{size}})-1} \text{OperationsPerLayer}(i) \\ &= \sum_{i=0}^{\lceil \log_2 V_{\text{size}} \rceil - 1} 2^i \\ &= 2^{\lceil \log_2 V_{\text{size}} \rceil} - 1 \\ &\geq V_{\text{size}} - 1 \end{aligned} \tag{3.3}$$

From this inequality it can be concluded that the number of cycles required by the straightforward implemented reduction tree is the same or even more than using a sequential algorithm that simply performs the $V_{\text{size}} - 1$ combinations required to reduce one vector.

From equation (3.3) it can be concluded that instead of mapping the reduction tree to the SIMD, it would be just as fast, or even faster, to implement a sequential type of algorithm. This is accomplished by shifting the elements to the CP and in parallel combine them one by one as they arrive. The pseudo code for this straightforward method is given in algorithm 1. In the pseudo code **right(x)** is used to indicate that element x is being read from the right neighbouring PE.

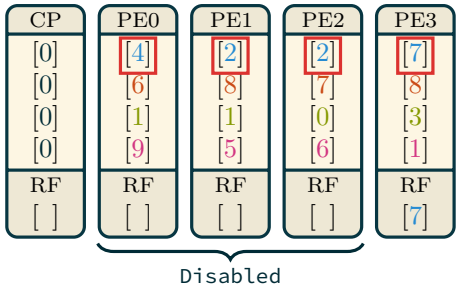
Algorithm 1 Straightforward Approach.

```
LoadAddr ← addressFirstVector
for i = 0 to N_vect do
  v ← load(loadAddr)
  for j = 0 to V_size do
    CP: combine(right(v))
    v ←right(v)
  end for
  LoadAddr ← LoadAddr + 1
end for
```

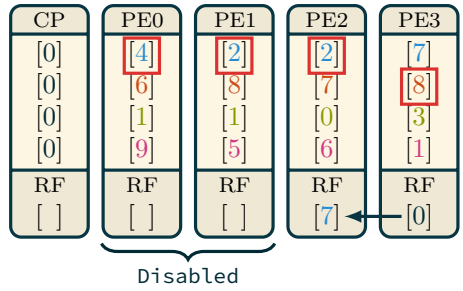
3.3.2 Pipelined Reduction

Since it is impossible to exploit the DLP within a single vector with a neighbourhood network as shown in the previous section, the parallelism has to be found elsewhere. In this section the novel *pipeline reduction* and *diagonal access reduction* algorithms are introduced that exploit parallelism in the number of vectors that have to be reduced. Using this parallelism the communication pattern is transformed such that only local transactions are required, and the whole PE array can perform combine operations on the input data.

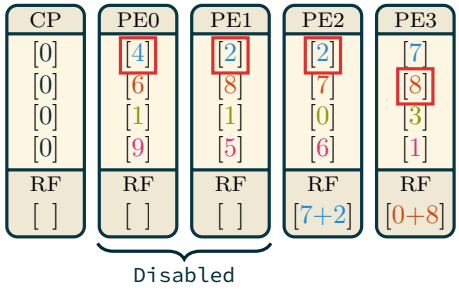
The pseudo code for the pipelined reduction algorithm is given in algorithm 2. The key of this algorithm is that it operates on multiple vectors in parallel, i.e., at any given moment in time all the PEs perform combine operations for different vectors. After a PE has performed a combine operation, the result is passed to the next PE. This PE will then load the element from its DMEM that corresponds to the vector of the received data, and repeat the process. For clarity a visualisation is given in figure 3.4.



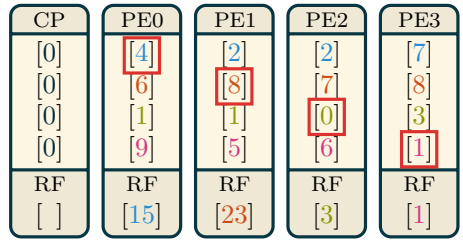
(a) Last PE loads top element. Rest of the PEs is disabled by predicating their instructions based on the ID of the PE.



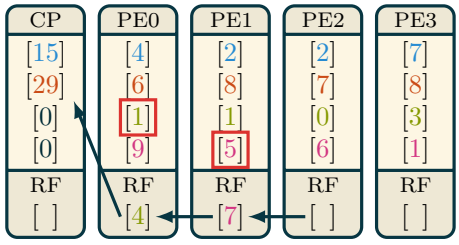
(b) Increase load address in active PEs, enable next PE and shift loaded value to the left.



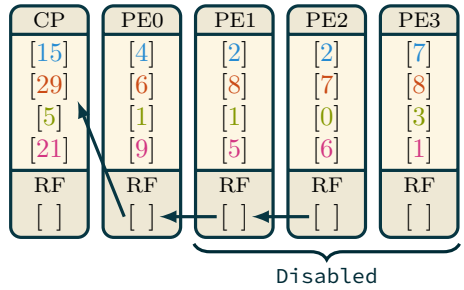
(c) Load next value and reduce it with the element just received.



(d) Repeat until all PEs are active. The 'pipeline' is now filled.



(e) When a PE is done with all vectors in its DMEM, disable it again.



(f) Repeat until all sums have ended up in the CP.

Figure 3.4: Visualisation of the pipelined reduction Algorithm. For this Figure *summation* is used as the combine operator.

Algorithm 2 Pipelined Reduction.

```
LoadAddr ← addressFirstVector
for i = 0 to N_vect + V_size − 1 do
  if (PEid ≥ V_size − i) and (loadAddr < endAddr) then
    v ← load(loadAddr)
    s ←combine(left(s), v)
    CP: store(left(s)) {if i > V_size − 1}
    loadAddr ← loadAddr + 1
  end if
end for
```

In this pipelined reduction algorithm, three phases can be recognized:

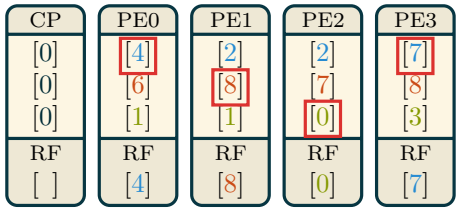
1. **Filling the pipeline:**
   In this phase not all PEs are active. It takes N_{PE} steps before PE0 receives its first element. This phase corresponds with figure 3.4a – 3.4c.

2. **Maximum occupancy:**
   If $N_{\text{vect}} \geq V_{\text{size}}$, then there will be a point where all the PEs are active. In this phase V_{size} PEs will perform a useful combine operation per step in the algorithm. See figure 3.4d.

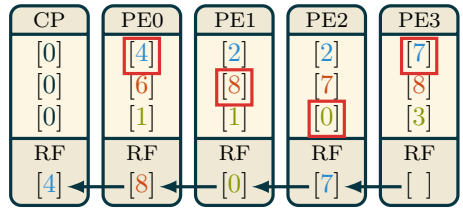
3. **Emptying the pipeline:**
   Once the last PE in the array has processed the last vector, it can be disabled. From this point on the remaining PEs will finish one by one until the first PE in the array completes. This corresponds with figure 3.4e – 3.4f.

3.3.3 Diagonal Access Reduction

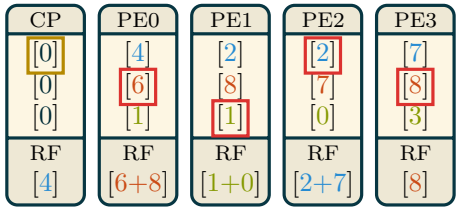
If $N_{\text{vect}} < V_{\text{size}}$, the pipelined reduction algorithm never enters the most efficient phase (phase 2). Therefore, if N_{vect} is much smaller than V_{size} it is better to take a different approach. By accessing the elements in a diagonal pattern from the start and using wrap-around, efficient reduction is possible for all situations where $N_{\text{vect}} \leq V_{\text{size}}$. The pseudo code for the diagonal access reduction algorithm is given in algorithm 3. A visualization is provided in figure 3.5.



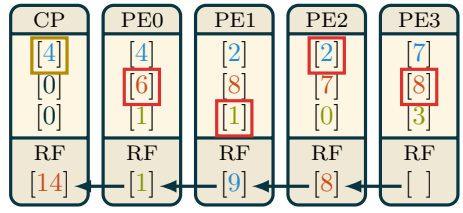
(a) Set load address to $\text{PE}_{ID} \mod N_{\text{vect}}$ (red squares) and load first element.



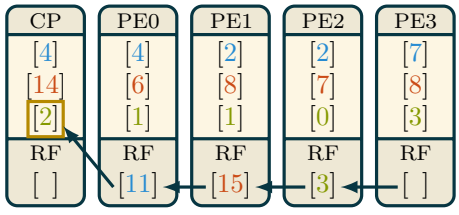
(b) Shift left.



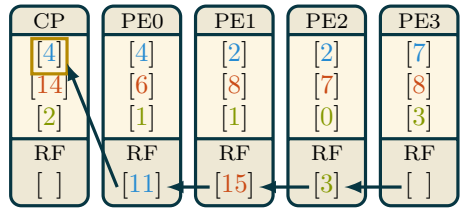
(c) Increase load address and use warp-around if required (PE2). Load element and combine with shifted value. CP: store received value.



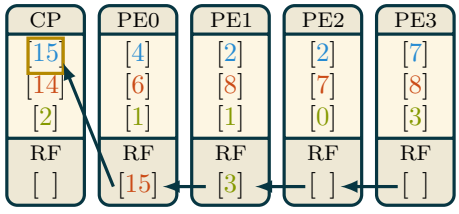
(d) Shift left.



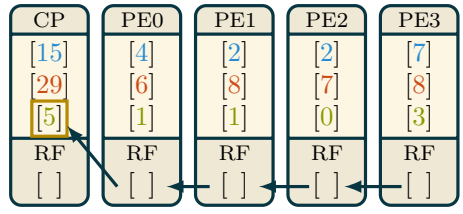
(e) Repeat until every input element is touched.



(f) Shift left. CP: combine incoming with stored elements.



(g) Shift left. CP: combine incoming with stored elements.



(h) Repeat until all elements are reduced.

Figure 3.5: Visualisation of Diagonal Access Reduction, again *summation* is chosen as the combine operation.

Algorithm 3 Diagonal Access Reduction ($N_{\text{vect}} < N_{\text{PE}}$).

```
LoadAddr ← addressFirstVector + (PE_ID mod N_vect)
s ← load(loadAddr)
for i = 0 to N_vect − 1 do
  loadAddr ← wrap(loadAddr + 1) {no modulo required!}
  v ← load(loadAddr)
  s ← combine( v + right(s))
end for
for i = 0 to V_size do
  s ← right(s)
  CP: combine(Result[i mod N_vect],right(s))
end for
```

3.4 Analysis and Evaluation

In this section the two novel reduction methods and the reference methods are analysed and evaluated in terms of running time, chip area and energy consumption.

First running times of the various approaches are obtained by using a cycle accurate simulator which is verified against register-transfer level (RTL) code. The *measured* running times are plotted as *continuous* lines in figure 3.6. The vector size (V_{size}) is fixed at 128 elements. Apart from the measured values, the specific constants for the complexity formulas of the algorithms are derived from the source code to approximate the running times for any combination of N_{vect} and V_{size} (equations (3.4) to (3.7)). To demonstrate the accuracy of the formulas, the approximated lines also plotted in figure 3.6.

$$\text{Straigthforward}(V_{\text{size}}, N_{\text{vect}}) = 10 + 12 \times N_{\text{vect}} + \frac{11}{8} \times V_{\text{size}} \times N_{\text{vect}} \tag{3.4}$$

$$\text{Pipelined}(V_{\text{size}}, N_{\text{vect}}) = 26 + V_{\text{size}} \times 4 + N_{\text{vect}} \times \frac{19}{8} \tag{3.5}$$

$$\begin{aligned} &\text{DiagonalAccess}(V_{\text{size}}, N_{\text{vect}}) = \\ &\quad 12 + 11 \times N_{\text{vect}} + \log_2 \frac{V_{\text{size}}}{N_{\text{vect}}} \times (37.5 + N_{\text{vect}}) + V_{\text{size}} \times 0.5 \end{aligned} \tag{3.6}$$

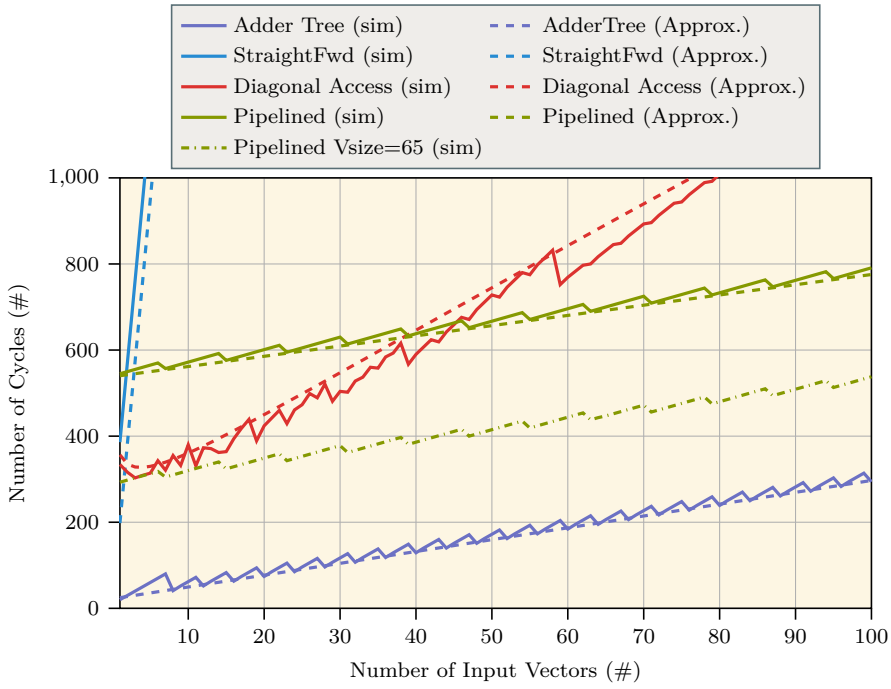


Figure 3.6: Measured and approximated running times of the various methods for varying number of vectors (N_{vect}) and fixed vector size ($V_{\text{size}} = 128$). The approximated lines are very close to the measured results. The largest deviations can be seen in the diagonal access algorithm, where a repeating irregularity is not captured in the approximation.

$$\begin{aligned} \text{adderTree}(N_{\text{vect}}, N_{\text{PE}}) &= 14 + \alpha + N_{\text{vect}} \times \left(\frac{6}{\alpha} + 2\right), \\ &\text{with } \alpha = \text{nextPowerOfTwo}\left(\lceil \log_2 N_{\text{PE}} \rceil\right) \end{aligned} \tag{3.7}$$

Since the adder tree requires exactly the same amount of cycles for $64 < V_{\text{size}} \leq 128$, the same line would hold for $V_{\text{size}} = 65$. The software approaches would however need to do less work and would thus finish faster. To illustrate this a purple line is added for the pipelined algorithm for $V_{\text{size}} = 65$. This line can thus be compared directly to line of the adder tree, indicating how much the performance difference can vary if V_{size} is between two consecutive powers of two.

As can be seen in figure 3.6 the *pipelined* and *Diagonal Access* algorithms provide an enormous speedup compared to the straightforward method for more than a couple of vectors. When analysing the runtime expressions for fixed vector size on both these algorithms, i.e., equation (3.5) and equation (3.6), it can be seen that both asymptotically converge to a runtime of $\mathcal{O}(N_{\text{vect}})$. However, the pipelined algorithm has a smaller constant in front of this term ($^{19}/_{8}$), whereas the Diagonal access algorithm comes in at 11. A factor 4.6× difference, which can also roughly be observed as the difference in slope between these algorithms in figure 3.6. This larger constant is caused by the more complex control overhead that is required by the Diagonal Access algorithm, which keeps coming back every cycle. The advantage is however that the Diagonal Access algorithm immediately engages all PEs. The pipelined approach on the other hand has a high initial cost as it fills the pipeline, and PEs are involved more gradually. But once filled, the recurring control complexity is much lower than that of the Diagonal Access algorithm. The results in the Diagonal Access algorithm outperforming the pipelined method for small N_{vect}, while for large N_{vect} the pipelined approach is favourable. Based on the cross-over point, a hybrid approach can be taken where based on N_{vect} dynamically the optimal algorithm is selected.

Interesting is that the running time of the adder tree grows at about the same rate as the pipelined reduction algorithm. In fact, as can be derived from the running time formulas, in the used implementation the pipelined reduction algorithm grows at about 2.38 cycles per vector while the adder tree grows at 2.75. At some point the software reduction would thus even be *faster* than the dedicated adder tree.

Table 3.1: Area Overhead, Running Time, Power and Energy comparison for the various approaches obtained by post-synthesis simulation. $N_{\text{vect}} = 100$ and $V_{\text{size}} = N_{\mathbf{pe}} = 128$.

Approach	**Area Overhead**	**Speed (cycles)**	**Power** (μW)	**Energy** (pJ)
Straightfwd.	0 %	18814	51	2398
Diagonal	0 %	1377	63	217
Pipelined	0 %	786	56	110
Adder Tree	6.5 %	294	82	60

This effect though is highly dependent on the target architecture. Both algorithms have the same complexity and are theoretically able to grow at a rate of one cycle per vector. In the selected target architecture however, one cycle is required to load the vector from memory, one to do the actual reduction, and the rest is control overhead shared over a number of vectors. An expansion of the target architecture with zero overhead loop support and dual issue PEs would enable a growth of only one cycle per vector for both the dedicated hardware, and the pipelined reduction method.

Table 3.1 shows the area overhead, and energy results for a fixed input size. These numbers are obtained by synthesizing the SIMD for 400 MHz with a commercial 40 nm library. Post synthesis simulation is used to obtain the power and energy results. As can be seen in the table, the adder tree consumes less energy, but these numbers are excluding memories. If the data memories are chosen to be 16-bit wide, 1 kB large and also built in 40 nm technology, the CACTI memory tool [149] calculates an access energy of 76×10^{-2} pJ. For the tested configurations this would result in an additional energy of 98×10^{2} pJ, making the energy difference between dedicated hardware and the novel algorithms negligible.

Case Study — Fast Focus on Structures

To evaluate the effectiveness of the novel reduction algorithms in a practical application, the Fast Focus on Structures application [53] was mapped to the target platform as a case study.

In the fast focus on structures (FFOS) algorithm the centres of organic light-emitting diodes (OLEDs) have to be detected from an image. In order to do so, reduction is used in two parts of the algorithm. Once to merge partial histograms and convert them to a cumulative histogram (CH) and cumulative intensive area

Table 3.2: Cycles times for both the adder tree and the novel reduction algorithms for FFOS on a 120×45 input image.

Application	Adder Tree	Novel Algorithm
CH/CIA calculation	2580	2540
Row Projection	379	970

(CIA), and once to obtain the sum of the rows of the image and detect peaks in that projection. More information on FFOS is provided in section 2.3.2.

The cycle counts for the various parts of the application with both the novel software techniques and a dedicated adder tree are given in table 3.2. As is shown in the table, for this practical example, the software reduction technique is even faster for the CH/CIA calculation. This is due to the flexibility of the software approach. Where the adder tree always gives its result directly to the CP, the software approach is able to do some post processing in parallel with the CP, reducing the running time. For row-projection, the detection of peaks in the row projection on the CP takes so much time that the reduction operations on the PEs are completely hidden. It is only the initial start-up cost that makes the software reduction technique slower here. Overall the FFOS application with dedicated hardware is only 7.1 % faster than with the software reduction techniques.

3.5 Related Work

Reduction is encountered frequently in the target domains of wide-SIMDs and multiple solutions to support reduction have been proposed in the past. The most common approach is to implement dedicated hardware to support a fixed type of reduction. For example Seo et al. [132] suggest a dedicated adder tree as an extension to AnySP [172] in order to support the H.264 video codec efficiently. Other examples of SIMDs optimized for video processing that include dedicated hardware include SIMD-2D [115] and the work by Li et al. [90].

In the SLiM-II [21] a dedicated interconnect is used to support reduction. Essentially the red lines in figure 3.3 are implemented as direct, one cycle latency, connections between PEs. To perform one reduction operation with this network $\lceil \log_2 V_{\text{size}} \rceil$ communication steps are required. This approach is flexible in type of operation, but a single reduction operation takes $\mathcal{O}(\lceil \log_2 V_{\text{size}} \rceil)$ operations, as consecutive operations cannot be pipelined. Furthermore implementing the red

lines as connections would result in PE_0 having $\lceil \log_2 V_{\text{size}} \rceil$ additional connections, which the instruction set must support.

It is clear that efficient reduction support for wide-SIMDs is a relevant topic for many applications. The proposed solutions in the related works all use additional hardware to support reduction causing them to either lose generality, or end up with an inherently slower and more complex design. The novel reduction algorithms introduced in this chapter avoid the downsides of dedicated hardware and offer an interesting trade-off between pure performance, flexibility, scalability, and chip area.

3.6 Conclusions

In this chapter two reduction algorithms are proposed which are optimized for highly scalable, low-power interconnects that provide only minimal connectivity. It has been shown that the algorithms are much more effective than a straightforward approach and can even compete with dedicated hardware solutions. The added flexibility of the algorithms can in practical cases give an edge over hardware solutions. Since there is no additional hardware involved and only short local wires for communication are required, these software approaches are cheaper in area and can scale virtually unlimited. As almost all types of interconnect provide the required connectivity, these algorithms can be mapped to existing processors that lack hardware support. For future designs it should be a reason to reconsider adding hardware support at all.

Chapter 4

Datawidth-Aware Multiplication

This chapter is based on the work published in "Datawidth-Aware Energy-Efficient Multipliers: A Case for Going Sign Magnitude" [155].

The multiplication operator is used in many application domains, including linear algebra, image/signal processing, and deep learning. Despite hardware support present in most modern cores, multiplication remains one of the most energy hungry arithmetic operations. The energy efficiency of current multipliers can be improved, however, by taking into account that the operands typically do not utilize the full width of the datapath. Many applications, such as quantized neural networks, rarely require the full datapath width of the machines they execute on [37], and could benefit tremendously from multipliers that have reduced energy consumption for narrow operands.

This chapter explores the inefficiencies in typical multipliers, and evaluates seven *datawith-aware* multiplier designs. Post-layout energy analysis is performed to obtain the energy efficiency of each design for a number of representative benchmarks targeting the consumer market. Results show a significant improvement in energy efficiency compared to a 32-bit Baugh-Wooley baseline multiplier. A 32-bit sign-magnitude based design, integrated in a two's complement datapath, is shown to have a 1.38 times better energy efficiency than a baseline two's complement multiplier. In the best case (JPEG encoding), the energy efficiency is increased by a factor 2.25, demonstrating that a sign-magnitude multiplier, and datawidth-aware multipliers in general, are an attractive option for ultra low-energy designs.

4.1 Introduction

Many application domains contain algorithms that heavily use the multiplication operator. Therefore, almost all real-world processors contain hardware multipliers, despite their complex circuits compared to many other arithmetic units. The high complexity of hardware multipliers, and the high frequency at which

multiplications occur, contribute to the high energy usage of multipliers relative to other hardware in a system. In particular the integer unit can represent from 10 % for high-end central processing units (CPUs) [43] to as much as 20-49 % for digital signal processor (DSP) type of processors [80], of the total energy usage. Therefore it is important to improve the efficiency of multiplications to achieve a high energy efficiency of the overall system.

Since multipliers have such a significant impact, already many techniques have been developed to improve their energy efficiency. To identify further optimisation opportunities, this chapter investigates in general the relation between operands and energy consumption of a standard multiplier. In particular the relation between effective operand width and energy consumption is shown to be a promising direction to improve the energy efficiency of multiplication. In image processing, e.g., pixels often are represented by only eight bits. Operations on these pixels typically do not increase the effective bit width of the data elements by much. When such an image processing application is executed on a 32/64-bit datapath, the upper bits of the datapath do not need to be involved in the majority of the operations.

In this chapter a cycle accurate simulator is used to obtain the real operand width distribution of six representative benchmarks. From these measurements it is concluded that the majority of real-world multiplications indeed have operands with an effective operand width much narrower than the full datapath width (section 4.2.2). Furthermore it is shown that the minimum energy required for such operations is significantly lower than full-width multiplications, although standard multipliers fail to capitalize on this (section 4.2.3).

After identification and classification of various opportunities to exploit this potential with datawidth-aware techniques, seven different multiplier designs are implemented in 40 nm technology. Post-layout energy analysis is performed to obtain the energy efficiency of each design. The results show a significant improvement in energy efficiency for various computing kernels.

To summarize, the key contributions contained in this chapter are:

1. The relation between selected operand properties and energy consumption as claimed by related work [36, 6] is qualitatively investigated.

2. The concept of *datawidth-aware computing* is validated by investigating the relation between operand width and energy consumption.

3. Identification, classification, and evaluation of different datawidth-aware techniques for multipliers using *post-layout energy estimation*.

4. Proposal to integrate a sign magnitude multiplier in a standard two's complement datapath, and demonstration that this design has a *superior energy efficiency*.

The remainder of this chapter is organised as follows. Section 4.2 analyses the viability of datawidth-aware multipliers, and investigates the relation between several multiplication operand properties and energy usage. Seven datawidth-aware multiplier designs are introduced in section 4.3. Energy and area of these designs are evaluated in section 4.4. Related work is discussed in section 4.5, and section 4.6 finally concludes this chapter.

4.2 Viability of Datawidth-Aware Multipliers

To improve upon the classic multiplier design is not trivial as much research has already gone into optimizing this important building block. Opportunities for improvement can be found in specific scenarios however, rather than improving the general case. Specific operands may inherently require less energy than typical hardware consumes [36, 6, 17, 18].

The relations between various operand properties and energy consumption are investigated in detail in section 4.2.1. Out of these, the most promising direction is shown to be the relation between operand width and energy consumption. The intuition behind this *datawidth-aware* computing is that operations on narrow operands can be completed with less energy than the same operation on wider operands. There are two criteria that need to be met if datawidth-aware techniques are to improve on the energy efficiency of standard designs:

1. Multiplications with operands that are narrower than the width of the full datapath frequently occur in realistic applications.

2. Standard hardware multipliers do not (fully) exploit multiplications with narrow operands to obtain a higher energy efficiency.

In section 4.2.2 and 4.2.3 respectively validate both these criteria, confirming the viability of datawidth-aware multipliers to improve energy efficiency.

4.2.1 Relation between Energy and Operand Properties

Related work suggests several relations between operand properties and energy consumption [36, 6]. These relations might provide hints for optimization of the multiplier circuit, hence this section investigates these claimed relations.

To determine the relation between the operand properties and energy usage, a $5b \times 5b$ baseline multiplier is constructed. The baseline design is an *energy efficient* two's complement (2C) multiplier. In most cases, 2C multiplier designs either follow the Baugh-Wooley algorithm [14] or the Booth algorithm [98]. Compared to a Booth-based multiplier, a Baugh-Wooley based multiplier of the same bit width typically has a higher energy efficient [140, 82]. Therefore a Baugh-Wooley based design is selected as the baseline (figure 4.7a).

The energy usage of a multiplication depends on two things, being the operands of the multiplication, but also the state the circuit is in at the start. For example, when the same operands are multiplied consecutively there will be no toggling in the circuit the second time, vastly reducing the consumed energy. To capture these relations every pair of $A \times B$ followed by $C \times D$ multiplications is simulated on this $5b \times 5b$ multiplier, after which post-layout energy estimation is used to determine the energy usage of $C \times D$.

Next, several operand features are handcrafted (table 4.1), and machine learning is used to assess their relation with energy consumption. In particular the following operand features are defined:

1. The hamming distance between consecutive operands A, C and B, D, as proposed by Fujino and Moshnyaga [36].

2. Ahn and Choi [6] propose to use the difference in sign bits of consecutive operands with the intuition that it is indicative for the difference in leading bits of the operands. Rather than using only the difference in sign bit, in this chapter the feature is extended to count the total number of matching leading bits, which is slightly stronger and does not rely on the intuition that many of the leading bits will be the same. Note: this would hold for many real world data, as demonstrated in section 4.2.2, but not for the exhaustive 5-bit experiment conducted here.

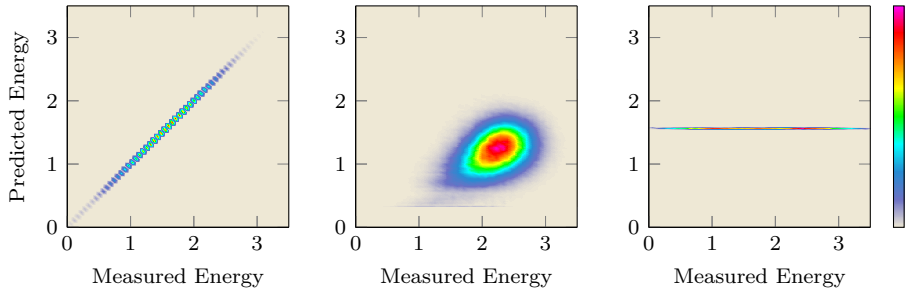
3. Finally the number of zeroes in the individual operands is used, following the intuition that numbers with many zeroes probably cause less carries and internal toggling.

Table 4.1: Correlation coefficient of individual operand features and complete models, and learned linear regression coefficients.

Feature	Correlation Coef.	Lin. Reg. Coef.
Hamming(A,C)	0.08	-0.02
Hamming(B,D)	0.22	0.38
Leading(A,C)	-0.14	-0.29
Leading(B,D)	-0.19	-0.17
Zeroes(A)	-0.09	-0.21
Zeroes(B)	-0.19	-0.44
Zeroes(C)	-0.16	-0.37
Zeroes(D)	-0.16	-0.37
Linear Regression	0.41	—

To qualify how indicative these features are for the energy consumption of the multiplier, the Pearson correlation coefficient is calculated between each of the features and the measured energy. This coefficient measures the linear correlation between two variables, and results in a number between -1 and 1. Here minus one represents perfect negative correlation, zero means no correlation, and 1 means perfect correlation. The correlation coefficient for each of the features is presented in table 4.1. The highest correlation is with the hamming distance between B and D, as might be expected based on the intuition presented by Fujino and Moshnyaga [36]. Surprising is however that the hamming distance between A and C has almost no correlation, which is most likely due to the asymmetric nature of multipliers. A similar, yet weaker, trend can be observed for the number of matching leading bits. Also here the B and D operands have higher correlation with the energy. Finally the number of zeroes feature proposed in this chapter shows reasonable correlation with the energy, although the strongest correlation remains with the hamming distance between B and D.

In order to find out if a combination of these features has a higher indicative value, linear regression is applied to the proposed features. The resulting correlation coefficient, i.e., 0.41, is much higher than the individual metrics, showing a combination of these features holds some predictive value for the energy usage. This is visualized in figure 4.1, where figure 4.1a is measured energy versus measured energy to resemble the perfect prediction, a diagonal line. Note that the colours in the figure indicate the number of measurements on a given location. The diagonal has the highest concentration of measurements in the centre, as the energy distribution essentially fits a normal curve. The energy predicted by the linear regression model is visualized in figure 4.1b. Here it becomes clear the



(a) Oracle predictor.

(b) Linear Regression on features of table 4.1.

(c) Linear Regression on operands A, B, C, D.

Figure 4.1: Measured versus predicted energy. Ideal is figure 4.1a. Linear regression on the proposed features is not very accurate, and appears to have a bias to predict too low in figure 4.1b. Linear regression directly on the input operands in figure 4.1c however mainly predicts the median energy, and ignores the input operands altogether, demonstrating that, although far from perfect, the proposed features do hold some predictive value.

predictive value of the model is not particularly high, as most points seem to cluster around a central area. The learned coefficients of each metric are given in the last column of the table. Interestingly it is not the hamming distance between B and D that has the highest absolute coefficient, but rather the number of zeroes appear to have a strong negative correlation with the energy consumed. The smallest coefficient (in absolute sense) is assigned to the hamming distance of A and C, confirming the observation that this feature has almost no influence.

The somewhat disappointing predictive value of the operand features raises the question if these features add anything over a linear regression on the operands directly. Direct linear regression over A, B, C, and D shows however that the model simply ends up predicting the average energy as shown in figure 4.1c. The coefficients assigned to A, B, C, and D are extremely small, as linear regression fails to find a direct linear correlation between the operands and energy consumption. This demonstrates the added value of the manually crafted features described in table 4.1.

The results presented in this subsection show some promise for further optimisation, in particular the number of zeroes appears to be a reasonable indicator of a low-energy operation. The energy models remain relatively inaccurate however. To encourage research in this direction, the exhaustive measurements (over one million post placement and route energy measurements) have been made freely

Table 4.2: Benchmarks by domains and type of operations.

	Coding	**Filtering**
Audio	MP3 Encode [86]	128-tap FIR Bandpass [1]
Graphics	JPEG Encode [73] H264 Encode [48]	Sobel Edge Detect [141] YUV to RGB conversion [28]

available [156], accompanied with the scripts to generate the results presented here.

4.2.2 Operand Width Distribution

The 'number of zeroes' features investigated in the previous subsection hints at opportunities for improving the energy efficiency of multipliers using datawidth-aware techniques, but it is far from conclusive. The next two sections investigate if the principle of datawidth-aware computing can be applied to multipliers. The first step is to investigate the operand with distribution in the real world, which is done in this subsection.

Related work on datawidth-aware computing assumes a distribution of operands where each bit has equal probability to be either a one or zero [180, 51]. This assumption might not hold for typical workloads. Therefore in this chapter the true operand width distribution is determined by tracing the multiplications that occur in real applications. Six benchmarks are selected from two important domains in consumer market devices: audio and graphics. Within these domains two common types of operations are selected, i.e., encoding and filtering (table 4.2). The coding benchmarks are based on the *consumer* section of the MiBench benchmark set v1 [47], but with updated versions of h.264 and the lame MP3 encoder.

To obtain traces of multiplications and their operands in the selected benchmarks, the cycle accurate simulator of OpenRISC [110] is used. The simulator is adapted to generate a trace of all the multiplications that occur in a benchmark. To avoid capturing multiplications from an operating system (OS), the benchmarks are executed bare-metal on the simulator. This procedure is illustrated in figure 4.2 in the *Multiplication Tracing* box.

For a 32 bit datapath like the OR1K architecture, all input operands of the multiplication are 32 bit 2C numbers. However, the effective data-width (EW) of

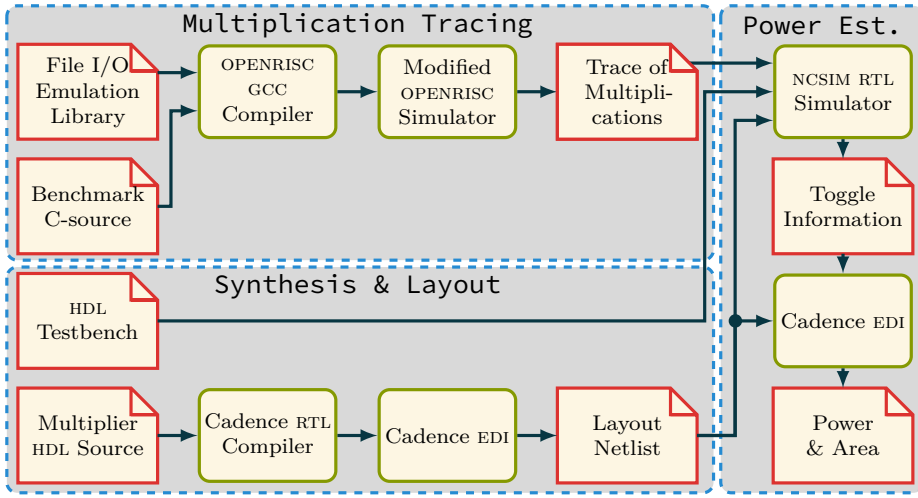


Figure 4.2: Tool flow for multiplication extraction and post-layout power/area estimation. The top part shows how multiplications operands are extracted from the benchmarks. The lower part details the hardware description language (HDL) complication. The part on the right shows how the results of both are combined to perform activity based power estimation on the synthesized netlist.

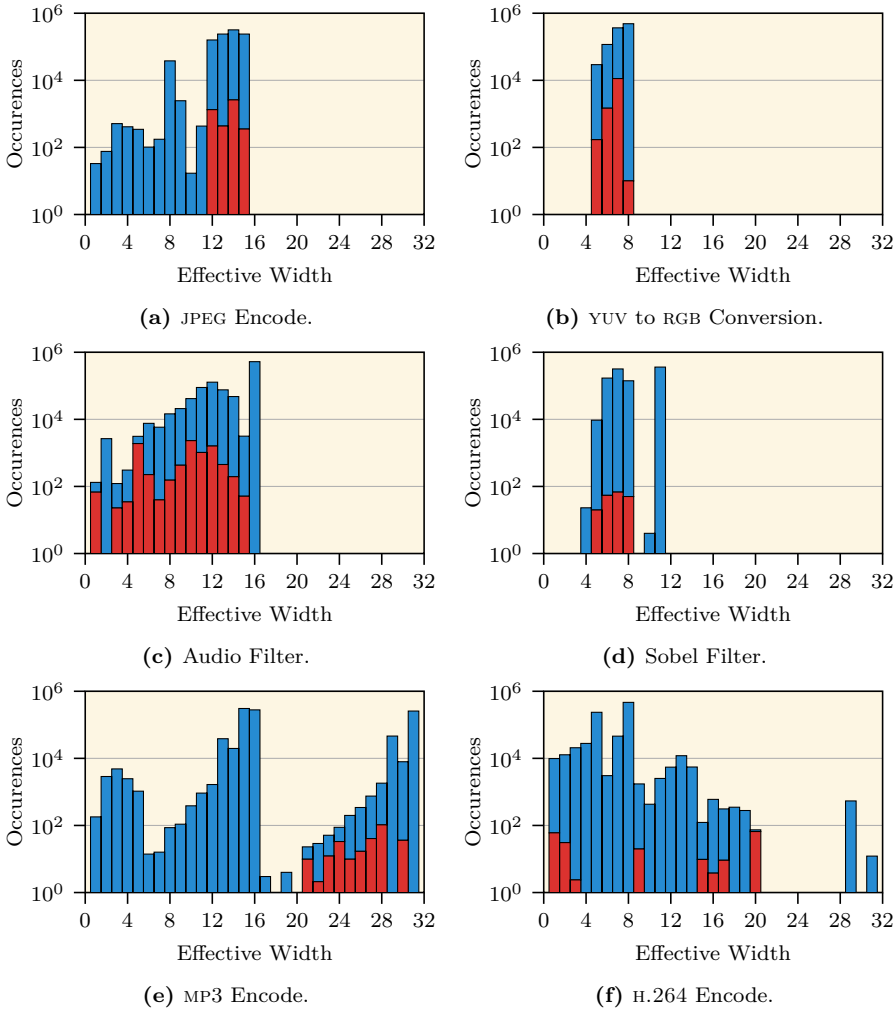
these operands is often less than 32-bit, were we define the EW of a 2C number as the minimum required number of bits to represent its absolute value in 2C notation. E.g., the effective datawidth of -5 is three (`101`). As an exception, the width of `0` is defined to be one. Since the largest operand of an operation presumably has the largest influence on the required energy for the operation, the maximum effective data-width (MEW) of the input operands is used to define the effective width of the entire operation. The MEW of a multiplication of two binary numbers N and M is defined as:

$$\text{MEW}(N, M) = \max\left(\text{EW}(N), \text{EW}(M)\right) \tag{4.1}$$

Using this definition and traces of the benchmarks, histograms of the MEW are constructed (figure 4.3). Note that using all traced multiplications for the power estimation is too time consuming, so they are subsampled by taking one thousand windows of one thousand consecutive multiplications. This is done to maintain the relation between consecutive multiplications while also covering multiple modes of the application. The histograms in figure 4.3 are based on the subsampled traces, although their shapes are virtually identical to the histograms over the full traces. The sampled versions are shown here because these represent the exact traces used to perform post-synthesis power estimation later in section 4.4.

From figure 4.3 it is clear that multiplications with a MEW smaller than the full width occur at a high frequency. In fact, multiplications with at least one operand that uses the full width are very rare, and are only encountered in MP3 and H.264 encoding (figure 4.3e-4.3f). These are the two benchmarks that use floats which are emulated on the standard OpenRISC architecture, which explains the higher MEWs for these benchmarks. The distribution of the MEW is not exponential, as a switching probability of 0.5 would lead to as was assumed by related work [180, 51]. In practice many applications only have operands with an EW of 16-bit or less. In general two peaks can be observed in the histograms: left of 8-bit, and left of 16-bit. This is to be expected, considering that the benchmarks are written in the C language, which provides data types of 8, 16, and 32-bit wide.

Overall it can be concluded that multiplications with operands that are narrower than the width of the full datapath occur frequently in real-world applications. Furthermore, there is a bias imposed by the programming language that favours multiplications with an MEW close to 8, 16, and 32-bit.



(a) JPEG Encode. **(b)** YUV to RGB Conversion.

(c) Audio Filter. **(d)** Sobel Filter.

(e) MP3 Encode. **(f)** H.264 Encode.

Figure 4.3: Maximum effective width distribution per benchmark. The height of the bars indicates the absolute occurrences on a logarithmic scale. The ratio (N.B., on a linear scale!) between the blue and the red parts of each bar represents the fraction of multiplications which involve at least one negative operand (red part) among all multiplications.

4.2.3 Relation between Energy and Operand Width

To relate the energy of a multiplication to its MEW, artificial inputs are constructed that have a MEW ranging from 1 to 31. The magnitude and sign of the operands are chosen randomly from the range dictated by the MEW. For each MEW value, one million multiplications are performed on a post-layout implementation of the baseline multiplier. To estimate the energy used by these multiplications, the tool flow as shown in figure 4.2 is used, with the exception that the synthetic operands with a fixed MEW are used as input, instead of the multiplication traces extracted from realistic applications. The baseline design, a 2C Baugh-Wooley multiplier, is synthesized for 100 MHz with a commercial 40 nm library. Both a 16-bit and a 32-bit version of the baseline design are tested as shown in figure 4.4.

As can be seen in figure 4.4, multiplications on randomized operands (the *baseline 16b/32b* trends) with a MEW of 16 or less can be performed with ~ 2.5 pJ on the 16-bit baseline multiplier. However, on the 32-bit baseline multiplier, the same multiplications use ~ 3.5 times more energy[1].

This proves that multiplications with a narrow MEW can be performed with less energy than those with a high MEW. However, the 2C baseline design is unable to exploit this. In fact, the energy cost of a multiplication is almost constant for all MEW values on a given multiplier design. Only for very low MEW values, the energy is somewhat less than a full width multiplication.

4.3 Datawidth-Aware Multiplier Designs

In the previous section it is shown that there is a significant potential for datawidth-aware designs to improve the energy efficiency of 2C multiplications. In this section, the domain of datawidth-aware computation is explored and various datawidth-aware techniques are classified. It is investigated how datawidth-aware techniques can be used to improve the energy efficiency of the baseline multiplier, and a sign magnitude (SM) multiplier integrated in a 2C datapath is proposed.

[1] As multipliers scale quadratically in logic with respect to the input operand size, operations with half the MEW require about 4 times less logic. Provided there is some additional logic surrounding the multiplier, such as input and/or output registers, which does not scale quadratically, achieving an energy improvement with a factor of 3.5 is realistic when the MEW is halved.

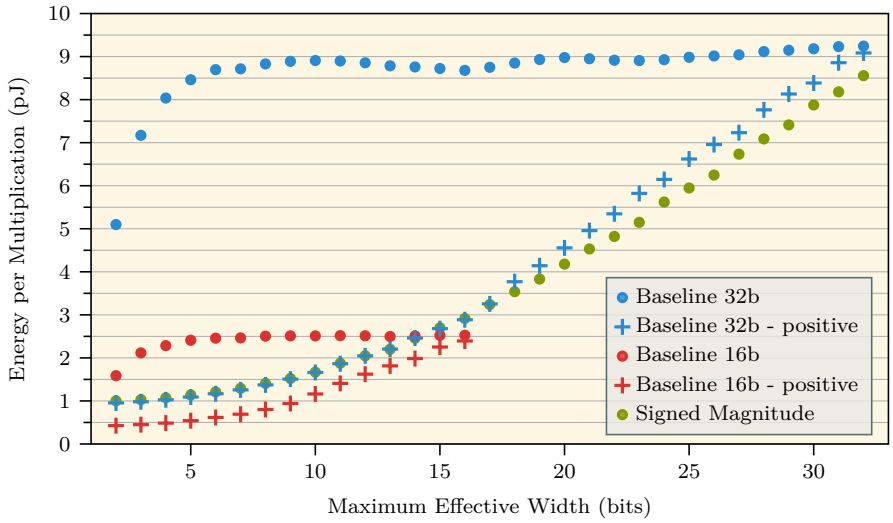


Figure 4.4: Relation between operand width and energy usage for standard two's complement multipliers (*Baseline* trends) and a sign magnitude multiplier. For mixed sign operands above a MEW of five, the energy per multiplication is almost constant for the baseline multipliers.

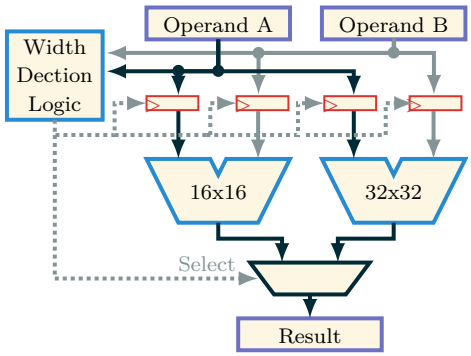


Figure 4.5: 32/16-bit mixed width multiplier.

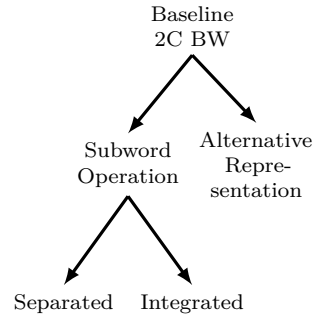


Figure 4.6: Classification of datawidth-aware techniques.

4.3.1 Subword mode — Separated

Based on figure 4.4, it is clear that on the 32-bit baseline multiplier, subword operations such as a multiplication with a MEW of 16, are supported very poorly energy-wise. A first approach to improve the 32-bit baseline design is to add better support for subword operations. In this chapter two methods based on this approach are considered.

One method is to extend the baseline multiplier by adding separate smaller multipliers that are optimised for the subword operations. In the extreme case a dedicated multiplier is added for every $n \times m$, for any number of bits n, m. Such a design quickly grows in area complexity, and is highly impractical. A slightly less extreme approach is to add a dedicated multiplier for each possible MEW ($n \times n$), which is exactly what is proposed by Bhardwaj et al. [15], in an attempt to find the theoretical optimum energy efficiency of a datawidth-aware design. In practice this design would consume a lot of area, and the overhead logic to select between the separate multipliers also consumes energy. Therefore, in this chapter, an even more practical version of this type of datawidth-aware multiplier is selected. The baseline 32-bit multiplier is extended with only one extra multiplier of either 16×16 or 8×8 (figure 4.5). The decision to support subwords of 16 or 8-bit based on the observation that there are peaks for these widths in the MEW histograms of the benchmark applications (figure 4.3). This type of datawidth-aware multiplier can be classified as *separated subword mode* multipliers. These designs correspond to the left most leaf in the datawidth-aware classification tree in figure 4.6.

4.3.2 Subword mode — Integrated

Using separate multipliers to improve the energy efficiency of subword operations is one method. However, this method incurs large area and leakage energy overheads. Another way of adding a subword mode to the baseline multiplier is to reuse the already available hardware, but isolate parts to prevent unnecessary switching. This is referred to as *integrated subword mode* (centre leaf in figure 4.6). Within the integrated subword class there are many possible designs. In this chapter four different designs are presented that implement a half-width mode.

Baseline Multiplier

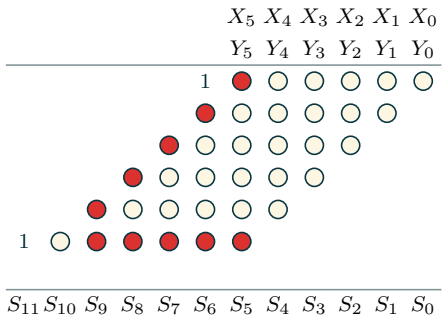
The baseline multiplier is a 32-bit, 2C, Baugh-Wooley signed multiplier, of which the partial-product bits have been reorganised according to Hatamian's scheme [52]. The layout of the partial products of the baseline is visualized in figure 4.7a. In the figure, a white dot represents the partial product of the corresponding bits of a multiplicand and a multiplier, while a red dot represents an inverted partial product bit. The designs with integrated subword mode are derived from this baseline design.

Least-Significant-Bit (LSB) Multiplier

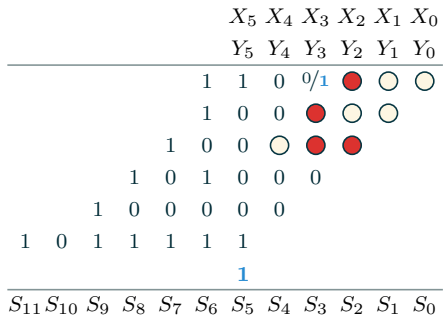
To enable half-width mode in a full-width multiplier, one option is to use the upper-right quadrant of the partial products, while isolating the rest of the multiplier logic. This design is referred to as the *Least-Significant-Bit* multiplier, and is depicted in figure 4.7b. The changes required to enable correct half-width calculation are highlighted in blue. When working in the default/full-width mode, the Least-Significant-Bit multiplier behaves exactly the same as the baseline reference. When both operands are detected as half width or less, half-width mode is engaged. The higher half of both operands, namely the extended sign bits, are *zeroed* to avoid unnecessary logic toggling.

Most-Significant-Bit (MSB) Multiplier

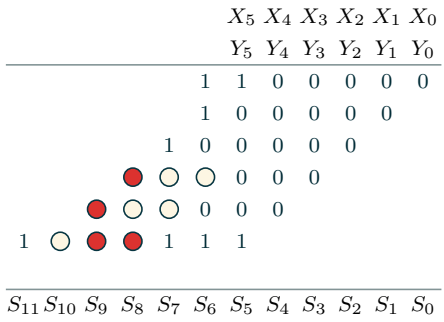
Instead of using the upper-right quadrant of the multiplier, the bottom-left quadrant can also be used to perform a half-width multiplication. This *Most-Significant-Bit* multiplier design is depicted in figure 4.7c. When half-width mode is triggered, the lower half of the input is routed to the upper half, and zeros are inserted into the lower half. Compared to the Least-Significant-Bit



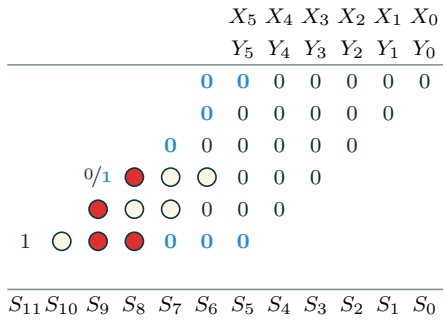
(a) Baseline multiplier.



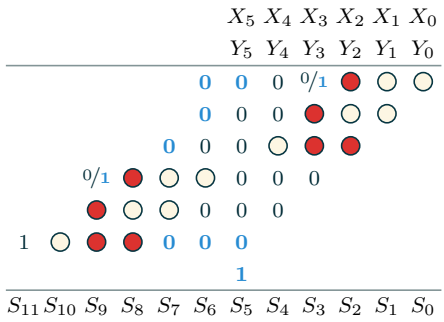
(b) LSB multiplier.



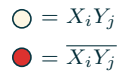
(c) MSB multiplier.



(d) Modified MSB multiplier.



(e) Twin multiplier.



(f) Legend.

Figure 4.7: Baseline and datawidth-aware multiplier designs. To simplify the view, partial products of 6×6 multiplier instances are shown. For the datawidth-aware designs (figure 4.7b-4.7e), only half-width mode is depicted. When working at full-width mode, they are equivalent to the baseline design (figure 4.7a).

multiplier, the Most-Significant-Bit multiplier requires a bit more hardware due to operand routing. However, the benefit of using the bottom left quadrant of the multiplier is that fewer partial products need to be disabled or modified.

Modified Most-Significant-Bit Multiplier

The *modified Most-Significant-Bit* multiplier is based on the Most-Significant-Bit multiplier. In the modified Most-Significant-Bit multiplier, inverted partial products are further *zeroed* (highlighted in blue) to reduce unnecessary toggling when the multiplier operates in half-width mode at the cost of a bit more control hardware. The design of the modified MSB multiplier is given in figure 4.7d.

Twin Multiplier

The *Twin* multiplier (figure 4.7e) is based on the work of Själander et al. [139], and is a combination of the Least-Significant-Bit and modified Most-Significant-Bit multipliers, enabling two half width multiplications in parallel, which is referred to as twin-mode. Dynamically enabling twin mode at runtime would require complex control logic, and is undesirable for an in-order processor. Therefore this is not considered for low-energy processors. The benchmarks used in this chapter have a *dynamic* mix of operand widths, which prevents the unconditional use of twin-mode. Therefore twin-mode is disabled in the experiments.

4.3.3 Alternative Data Representation

From the baseline trends in figure 4.4 it is concluded that the energy usage does not scale with the MEW for regular 2C hardware. In the previous sections, various techniques are introduced which potentially add better support for 2C multiplications with a small MEW. However, they all use the traditional 2C multiplication algorithm. In this section, it is shown *why* the energy usage of traditional 2C multiplication does not scale, and a design based on signed magnitude notation is proposed.

An important observation to be made is that in two's complement notation, negative numbers have leading ones, and positive numbers have leading zeroes. Thus, if the input of a multiplier changes sign between consecutive multiplications, all the leading bits (all bit positions above the EW of the operand) of the multiplier's input toggle. The hypothesis is that in this way even operands with a small EW can cause a lot of switching, increasing energy usage. This is verified by testing with only positive operands of different MEWs, such that the sign bits

Table 4.3: Energy usage breakdown (pJ/*op*) of the baseline multipliers with mixed and positive-only inputs at an MEW of 16.

	Sequential	Combinational	Clock	Total
Baseline 32b	1.26	7.19	0.23	8.68
Baseline 32b positive	0.88	1.78	0.23	2.99
Baseline 16b	0.54	1.90	0.07	2.53
Baseline 16b positive	0.54	1.79	0.07	2.39

are static. The results are shown in the *baseline positive* trends figure 4.4. The trends show that when only positive operands are used, the energy per operation scales much better with the MEW.

The *baseline positive* trends show that at a MEW of 16, the 32-bit multiplier is almost as efficient as the dedicated 16-bit multiplier. The energy breakdowns of this interesting point at a MEW of 16 are given in table 4.3. It can be seen in the table that for positive inputs the 16-bit and 32-bit baseline require about the same amount of energy. There is still a small energy gap which is attributed to the energy spent on the clock and the sequential logic (input and output registers). For the mixed inputs however, the energy consumption of the combinatorial parts of the 32-bit baseline is much higher than that of the 16-bit baseline. The results in table 4.3 and figure 4.4 confirm that the MEW-insensitive energy consumption of the baseline design can be fully attributed to the toggling of the leading sign bits.

Sign-Magnitude Multiplier With Conversion

The inefficiency of 2C can be mitigated by the use of a different data representation, such as sign magnitude (SM) notation. SM only uses one bit for the sign, and the remainder of the bits is used to represent the magnitude. Since the resulting magnitude is the multiplication of the, by definition always positive, input magnitudes, a SM multiplier should not suffer from the same inefficiency as a 2C multiplier. As an additional benefit the hardware of a SM multiplier is in itself simpler than that of a 2C multiplier. The sign can be computed with a single XOR gate, and the resulting magnitude can be found using an unsigned multiplier of size $(n-1)\times(n-1)$.

These claims are supported by the energy-MEW relation of a sign magnitude multiplier, which is shown in figure 4.4. For a mix of positive and negative numbers the curve follows the line of the positive-only 32-bit baseline, proving

Table 4.4: Corner cases for multiplication $A \times B = C$.

A	B	$\|C\|$
-(2^{N-1})	b	$\|b\| << (N\text{-}1)$
a	-(2^{N-1})	$\|a\| << (N\text{-}1)$
-(2^{N-1})	-(2^{N-1})	2^{2N-2}

that SM notation does not have the same inefficiency as 2C notation. The amount of energy used by the input and output registers is about equal for both the SM multiplier and the 32-bit baseline. Therefore, at very small MEWs, where the input and output registers are the dominant source of energy usage, the energy of the SM multiplier is about equal to that of the positive-only baseline. When the MEW increases, the advantage of the smaller and simpler magnitude multiplication starts to pay off, which is why the SM multiplier uses even lower energy at a MEW of 32-bit than the baseline 2C multiplier.

The energy advantages of using SM for multiplication are clear. However, as practical processors predominantly use a 2C datapath, a pure SM multiplier is of not much use. Therefore a design is proposed that integrates a SM multiplier in a 2C datapath, with a relatively small overhead.

A complication of such a design is the range mismatch between SM and 2C. The range of an N-bit 2C number is defined as [-(2^{N-1}), ..., 2^{N-1}-1], while the range of SM is [-(2^{N-1})+1, ..., -0, +0, ..., 2^{N-1}-1]. This mismatch causes three corner cases for the multiplication of $A \times B = C$, where either A, B, or both are the most negative value representable in 2C as shown in table 4.4. These corner cases can all be dealt with efficiently, by detecting them and exploiting the fact that the result is either 2^{2N-2}, or can be computed by shifting by N-1 bit positions. The full design of the sign-magnitude multiplier, including hardware for conversion and corner case handling, is shown in figure 4.8.

4.4 Evaluation

In this section, the various presented multiplier designs are evaluated for their area and energy efficiency using the tool flow shown in figure 4.2. The designs were mapped to 40 nm technology, and post-layout area and energy estimation was performed.

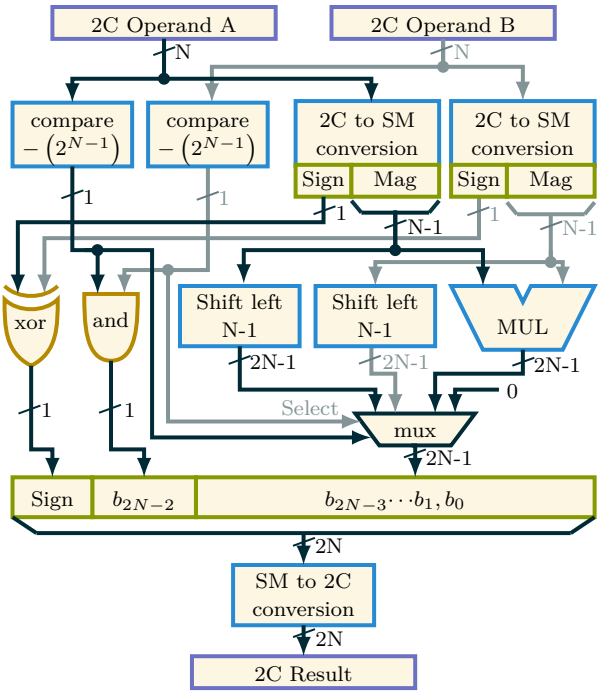


Figure 4.8: SM multiplier with additional logic to enable integration in a 2C datapath.

Table 4.5: Area overhead of the datawidth-aware multipliers.

Multiplier Design	Area Overhead (%)
Mixed Width 32/8	14.7
MSB	2.18
Modified MSB	8.04
Twin	5.61
Mixed Width 32/16	33.0
LSB	1.50
Sign-Magnitude	12.3
Area of the 32-bit Baseline	6210 µm^2

The area reported by the Cadence EDI tool for the layout of the 32-bit baseline multiplier in 40 nm technology is 6210 µm^2. For the other designs, the area overhead with respect to the baseline is given in table 4.5. As can be seen, the baseline occupies the smallest area. This is expected since the other designs are extensions of the baseline design. In particular the sign magnitude design requires more area as it requires extra hardware to fit into a 2C datapath.

In order to evaluate the real-world energy efficiency of the datawidth-aware multiplier designs the benchmark applications listed in (table 4.2) are used. The multiplications in these applications, which were extracted with the modified OR1K simulator and subsampled as described in section 4.2.2, are used as the inputs for post-layout energy estimation. The estimated energy efficiency of each application-multiplier pair is given in figure 4.9, and additionally the overall energy efficiencies of the multipliers is given in table 4.6. The presented energy numbers include detection logic which selects between full and half width nodes where necessary.

4.4.1 General Observations

The effectiveness of the datawidth-aware designs varies per benchmark, as can be seen in figure 4.9. The effectiveness is correlated with the histograms of the MEWs shown in figure 4.3. With the exception of the sign magnitude multiplier, the datawidth aware designs only improve energy efficiency if the MEW of a multiplication is less than, or equal to, half-width (or quarter-width for the Mixed Width 32/8 design). The MP3 encode benchmark in particular contains many multiplications with a high MEW (figure 4.3e). Consequently, the datawidth-aware designs have the lowest energy efficiency for this benchmark. Each datawidth-

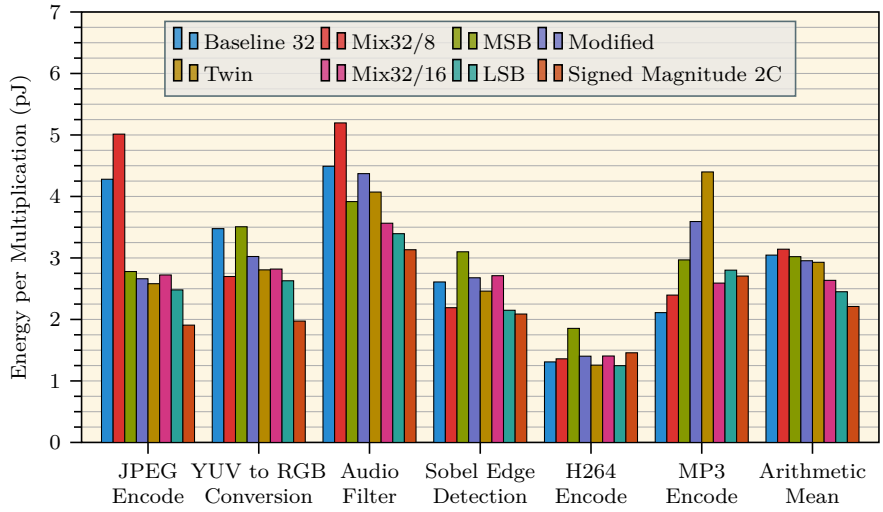


Figure 4.9: Energy efficiency of the various multiplier designs. The Sign-Magnitude multiplier has the lowest energy per multiplication for all benchmarks except h.264 and MP3 encoding. These applications also have high activity in the high MEW region (figure 4.3), explaining why in these cases the gains are less, and the overhead of the Sign-Magnitude integration can not be compensated sufficiently.

Table 4.6: The overall energy efficiency over all benchmarks.

Multiplier Design	**Energy per Op.** $\left(\frac{\text{pJ}}{op}\right)$	**Energy Eff.** $\left(\frac{op}{\text{nJ}}\right)$	**Improvement Factor** (×)
Mixed Width 32/8	3.14	319	0.97
MSB	3.02	331	1.01
Modified MSB	2.95	338	1.03
Twin	2.93	341	1.04
Mixed Width 32/16	2.64	379	1.16
LSB	2.45	408	1.24
Sign-Magnitude	2.21	452	1.38
Baseline	3.05	328	1.00

aware design introduces some extra logic which consumes energy. The design can only improve the efficiency when this overhead is compensated by enough multiplications with smaller MEW. For the MP3 encode benchmark no design overcomes the overhead of the extra hardware, which shows datawidth-aware designs are not always better than straightforward 2C multiplication.

The MEW histograms are only part of the story. Specifically, the datawidth-aware designs also perform poorly for the H.264 benchmark, while the number of multiplications with a high MEW (figure 4.3f) is not as high as in MP3 encode. The datawidth-aware designs profit most when the inefficiency of 2C notation, as described in section 4.3.3, is triggered by a mix of positive and negative operands. As can be seen in figure 4.3, where the ratio between the blue and orange bars indicates the ratio of multiplications with only positive operands and multiplications with at least one negative operand, most applications have such a mix of positive and negative operands. The exception is H.264, which has significantly fewer negative operands (figure 4.3f). Therefore, the datawidth-aware designs do not improve the efficiency as much as in the other cases.

In conclusion, the datawidth-aware designs benefit most when there are many operations with small MEWs, and a mix of positive and negative operands, which is the case for the benchmarks which are integer based.

Some peculiarities remain however. In particular, the overall energy usage on the baseline can not be predicted solely based on the MEW distributions. E.g., MP3 Encode runs much more efficiently on the baseline than JPEG Encode (figure 4.9), even though the MEW distribution of JPEG Encode shows mainly small MEWs. The cause is likely to be found in the relation between consecutive operands. When multiplications are executed on the same hardware, an earlier multiplication leaves the circuit in a state, which influences the energy consumption of the current multiplication. Exactly how consecutive operands influence the energy usage for multipliers is not known. Related work suggests different heuristics to estimate the energy consumption, such as the hamming distance[36], or the sign of the operands[6]. The accuracy of these heuristics is however not measured, and their application in our setting did not provide any additional insight, indicating further research into this relationship is required.

4.4.2 Subword — Separated

The designs that support subword modes by adding dedicated hardware for smaller MEW multiplications have the largest area overhead, with 14.7 % and

33.0 % for the 32/8 and 32/16-bit designs respectively. Which of the mixed multipliers is more efficient depends on the MEW histograms. Unsurprisingly the 32/8-bit has an advantage when many MEWs are below 8, such as the YUV to RGB benchmark, and the Sobel filter (figure 4.3b figure 4.3d). For the other benchmarks the 16/32-bit counterpart is more efficient. Compared to other datawidth-aware multipliers, the mixed multipliers are still attractive candidates for energy-efficient computing, although their major drawback is the relatively large area overhead compared to the other datawidth aware designs.

4.4.3 Subword — Integrated

The designs with integrated subword support score much better on area than the designs with separate dedicated hardware. The area overhead of the Least-Significant-Bit and Most-Significant-Bit multipliers is only 1.50−2.18 %. The modified Most-Significant-Bit design adds some extra isolation logic to the Most-Significant-Bit design in order to lower the energy per operation from 3.02 to 2.95 $\text{PJ}/_{\text{OP}}$, at the penalty of increasing the area overhead to 8.04 %. However, the Least-Significant-Bit design has the lowest area overhead and the highest energy efficiency of all the designs in this class, making it the preferred design. For colour conversion and the audio filter, the Least-Significant-Bit multiplier increases the energy efficiency by a factor greater than 1.3 compared to the baseline, while in the JPEG encode application, the efficiency is even improved 1.7×.

The twin multiplier is a combination of the Least-Significant-Bit and modified Most-Significant-Bit multipliers. Since the width of the operands is detected at runtime, twin mode is not used in this chapter. Unsurprisingly, the area overhead of the twin multiplier is larger than that of the Least-Significant-Bit design, and because of the extra logic it is unable to improve on the Least-Significant-Bit multiplier in the experiments. However, when a compiler could insert vector instructions such that twin mode becomes feasible, it has the potential to have a higher energy efficiency than the Least-Significant-Bit design. Therefore, the use of a twin multiplier can be an interesting option if half-width vector operations can be supported by software.

4.4.4 Sign Magnitude

The area overhead of the sign-magnitude multiplier is high at 12.3 % compared to the integrated subword designs. This is caused by the extra logic required

Table 4.7: Energy overhead for the SM multiplier with conversion.

Benchmark	Overhead (%)
MP3 Encode	21.0
JPEG Encode	19.2
H.264 Encode	19.8
FIR	27.5
Sobel	38.0
YUV to RGB	26.7
Average	25.4

to convert between 2C and SM. In itself, the sign-magnitude multiplier requires less logic than the baseline, since only an (n-1)×(n-1) unsigned multiplier is required. The sign-magnitude multiplier is inherently datawidth-aware, and can benefit at each EW. Even with the extra logic for the integration in a 2C datapath, it has the highest energy efficiency overall ($452 \cdot 10^9$ OP/J), as listed in table 4.6. The actual energy overhead for the conversions between 2C and SM as listed in table 4.7 is measured by comparing a pure SM multiplier with the sign-magnitude multiplier design proposed in section 4.3. On average the added conversion logic consumes 25.4 % extra energy compared to a pure SM multiplier, which is quite a lot, yet overall is compensated by the benefit of the datawidth effect. For JPEG encoding the energy efficiency is even increased by a factor 2.25, demonstrating the significant amount of energy that can be saved by a datawidth-aware multiplier despite the added overhead for conversion.

4.5 Related Work

Multiplication has always been an important operator in computing systems. In this section, an overview of the most important works of datawidth-aware computation, and in particular datawidth-aware multiplication, is given.

Power awareness is quantified in the work of Bhardwaj et al. [15], by constructing a theoretically optimal power-aware system. In their work, Bhardwaj et al. define power-awareness as a scenario-aware system, which is able to minimize energy usage based on different scenarios. Specifically, these different scenarios are defined as the datawidth of the input operands. The work sets a good framework for designing datawidth-aware systems, and inspired the design of the more practical mixed multipliers evaluated in our work.

Another important work that demonstrates the potential of datawidth-aware design is carried out by Brooks et al. [17, 18]. Based on the observation that over half of the integer operations require 16-bit or less across the SPECint95 benchmarks [142], a design which latches the input operands into segmented latches is proposed, thereby allowing to control which part of the inputs gets updated.

As discussed in section 4.3, the sign-magnitude representation is inherently datawidth-aware. This feature of SM representation is partially used to realize a low power multiplier by Zheng and Albicki [180]. In their work the multiplicand is converted to sign-magnitude representation, while the multiplier is kept in two's complement. The rationale behind this is that by keeping the multiplier in two's complement, booth recoding can still be used, while the generation of the negative partial products requires lower energy than with two's complement. This use of sign magnitude differs from our design presented in section 4.3, where both operands are converted, and no booth recoding is used, but the multiplication as a whole is performed in sign magnitude. The energy analysis presented by Zheng and Albicki are based on the estimated switching activity (ESA). Their assumption is that every bit has equal probability of being one or zero. This does not hold for real-world workloads as is demonstrated in this chapter in section 4.2.

The earlier discussed work of Bhardwaj et al. [15] states that a datawidth-aware design can be built by selecting the optimal hardware for each scenario/datawidth and constructing the final design as a combination of those elements. Although such a theoretical system is convenient to reason about the maximum energy efficiency, in order to get to a practical design, only certain scenarios should be selected. A good example of a datawidth-aware system, which is such a practical composition, is the twin multiplier by Själander et al. [139]. The twin multiplier can perform either one full-width multiplication, or two half-width multiplications in parallel. The power overhead for adding a twin-mode to a regular multiplier is shown to be only marginal for full width multiplications, while the power gains for half-width multiplications are substantial. This design is therefore an attractive candidate for a datawidth-aware, energy-efficient multiplier. As an extension to the work of Själander, the energy efficiency of this twin-multiplier is evaluated using post-layout simulation. Additionally the comparison with other datawidth-aware multipliers presented here shows that the twin design is an attractive design. However, some form of compiler support or offline analysis is required to effectively exploit twin mode.

Somewhat similar to the approach of Själander et al. is the work of Garofalo et al. [37]. In their work Garofalo et al. specifically re-target a RISC-V core for quantized neural networks. To efficiently support operations on small operand sizes, they introduce dedicated narrow 2-bit and 4-bit wide vector units, next to the existing 8-bit and 16-bit single instruction multiple data (SIMD) lanes. As in the work of Själander et al., compiler support is required to statically select the appropriate units. Furthermore, instead of (partly) reusing the existing wider hardware available, dedicated narrow units are instantiated, much like the mixed precision designs presented in this work. For this particular use-case, the design by Själander et al. may have provided area benefits not explored by Garofalo et al. The fundamental difference with the design proposed in our work, however, is the inability to dynamically profit from narrow operands, requiring dedicated compiler support and vectorisation.

An alternative datawidth-aware approach is proposed by Fujino and Moshnyaga [36]. Instead of defining the different possible datawidths as the scenarios to optimise for, the hamming distance between the current and previous input is exploited to improve energy efficiency. The idea is that if the hamming distance between the previous and current inputs is large, a significant amount of switching activity will occur in the multiplier. When the hamming distance is larger than a threshold, the input operands are dynamically transformed to a different representation in an attempt to reduce the energy usage. A possible downside of this approach is the requirement of a hardware hamming distance calculator, which may use a substantial amount of energy. Furthermore our analysis section 4.2.1 shows that although the hamming distance is the best single feature, it in fact not a very good indicator for the energy consumption with a correlation coefficient of only 0.22. Additional research into more predictive operand features might help improve upon the work of Fujino and Moshnyaga [36].

Another approach to enhance the energy efficiency by manipulating the input operands is proposed by Ahn and Choi [6]. Instead of using hamming distance as an indicator for the energy usage of a multiplication, Ahn and Choi recognize that many bits tend to have the same value as the sign bit in two's complement notation. Therefore, the sign bit can be used as an indicator for the most significant bits (MSBs). Based on the sign of the previous and current inputs, swapping the operands is argued to be beneficial for reducing the internal switching activity. The method proposed by Ahn and Choi is more suitable for real implementation than the method proposed by Fujino and Moshnyaga due to the lower logic complexity of the indicator. The analysis provided in section 4.2.1 shows however that the difference in sign bits, or even the number of equal

leading bits, is not a very good indication for energy consumption. Although this method addresses the same 2C inefficiency as discussed in section 4.3.3, it is not a datawidth-aware approach.

4.6 Conclusions

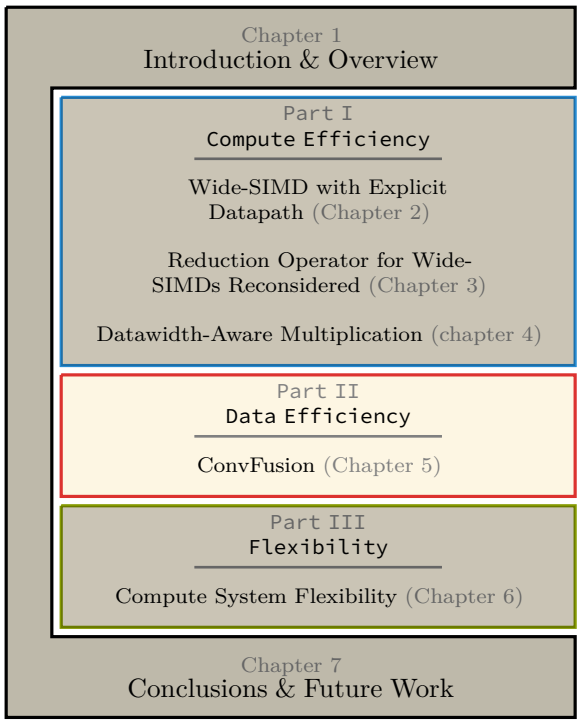
In this chapter the validity of using datawidth-aware techniques to improve the energy efficiency of hardware multipliers is proven. It is shown that multiplications with small operations occur frequently in real-world applications, but traditional multiplier designs fail to capitalize on this.

The opportunities of datawidth-aware techniques are explored by detailed analysis of different datawidth-aware designs, as well as characterization of six real-world applications. Based on this analysis, three main datawidth-aware techniques are identified with the potential to increase energy efficiency.

In this chapter the domain of datawidth-aware techniques is classified based on the identified techniques. To quantify the potential of datawidth-aware techniques, seven designs are selected from these classes, including a novel design that uses signed magnitude internally. Each design is fully implemented up to the layout phase, and analysed for its area and energy efficiency on six real-world benchmarks. A comparison is provided with a baseline 32-bit 2C multiplier.

It is found that datawidth-aware design is a promising approach to increase the energy efficiency of hardware multipliers. In particular the energy efficiency is improved by 38 % on average compared to the baseline when a sign-magnitude multiplier integrated into a 2C datapath is used, despite the 25 % energy overhead introduced by the conversion logic. In the best case (JPEG encoding) the energy efficiency is even increased by a factor 2.25. Due to integration in a 2C datapath the sign-magnitude multiplier has an area overhead of 12.3 %. When area is a concern the presented Least-Significant-Bit multiplier is an attractive alternative. With an area overhead of merely 1.5 % compared to the baseline, it is still able to improve the energy efficiency by 24 % on average.

Part II
Data Efficiency



Chapter 5

ConvFusion

This chapter is based on the work published in "Automatic Memory-Efficient Scheduling of CNNs" [161] and "ConvFusion: A Model for Layer Fusion in Convolutional Neural Networks" [162].

The superior accuracy and appealing universality of convolutional neural networks (CNNs) as a generic algorithm for many classification tasks have made the design of energy efficient CNN accelerators an important topic in both academia and industry. Of particular interest in the design and use of CNN accelerators is the scheduling of the computational workload, which can have a major impact on the quality of the final design. The many inherently independent operations in CNNs result in a vast scheduling space however, rendering the selection of the optimal schedule(s) non-trivial. To aid in this complex task, this chapter introduces a generic mathematical cost model of the external memory accesses, internal memory footprint, and compute load for CNN execution schedules. The model enables fast exploration of the scheduling space, including loop tiling, loop reordering, explicit data transfer scheduling, recomputation, and, crucially, layer fusion, which recently has attracted interest as a method to reduce external memory accesses. An accompanying open source tool is released to perform schedule space exploration for CNNs using the introduced cost model. Leveraging the code generation capabilities of this tool the proposed model is validated on six real world networks, demonstrating that layer fusion can reduce the external memory accesses by more than two orders of magnitude compared to the best non-fused schedules. Confusing at first glance however, a high-level energy analysis shows that the practical benefits of layer fusion may be overestimated if other parts of the system are not tuned accordingly.

5.1 Introduction

There is no longer any debate regarding the advantages of the class of convolutional neural network (CNN) algorithms. Many important problems, previously

deemed difficult if not impossible to compute, are now being solved by CNNs. The plethora of application domains includes: control systems, pattern recognition, power systems, robotics, forecasting, manufacturing, art, and medical diagnosis [4].

Despite their successful application to many computational problems over the last decade, CNNs also have several major drawbacks. In particular, they are both compute and memory intensive algorithms. In the early years this kept the execution of CNNs confined to data centres, as evaluation on available general purpose, embedded processors required too much energy to be practical in mobile, energy constrained devices. To overcome this, many dedicated CNN accelerators have been proposed since to bring CNNs to the edge, and in general reduce CNN energy consumption [127]. In modern technology nodes the main challenge in achieving a high energy efficiency for such accelerators is not so much the compute complexity, but rather the required memory accesses. Compared to an ALU operation, accessing an static random-access memory (SRAM) requires about 5× the energy, and going to external dynamic random-access memory (DRAM) about 200× [124]. This phenomenon, commonly referred to as the memory wall [177], will only aggravate with further technology scaling. To attain high energy efficiency it is therefore imperative that compute devices use their memory systems optimally.

Apart from techniques at the algorithmic level to reduce the total required memory accesses, minimizing the energy spent on the memory system constitutes of maximizing data reuse captured in small local memories. In essence this reduces the problem to finding a beneficial execution schedule for a given CNN. Due to the massive amount of independent operations in CNNs many valid schedules exist however, and finding the optimal schedule(s) for a given network and compute platform is exceedingly complex. In particular, the combination and parametrisation of scheduling techniques such as loop tiling [22], loop reordering [92], and more recently loop/layer fusion [7, 91, 165, 40], results in a vast scheduling space. To deal with this vast space existing research typically restricts itself to a subset of the complete space. This leads to the selection of potentially suboptimal schedules, and prohibits the generic application of the obtained results across different compute platforms.

This chapter introduces a generic cost model which can efficiently compute the cost of a CNN schedule in terms of external memory accesses, required internal buffer space, and total multiply accumulates (MACs). The model is platform agnostic, and capable of handling CNN schedules that can be created using loop

tiling, loop reordering, explicit scheduling of memory transfers, and layer fusion. This enables a fast search through possible schedules for CNNs without the need to perform profiling runs to obtain the cost of a particular schedule. The model is generic, and as such can be integrated in auto-schedulers for various accelerators and architectures by adequately bounding the schedule space based on specific architectural properties. A proof of concept, open source tool is developed building upon Keras/TensorFlow [23] as a front-end, and Halide [124] as a back-end. This tool is capable of performing exhaustive design space exploration for selected CNNs using the proposed model. Furthermore the Halide back-end generates code for each schedule, and instruments that code such that the modelled costs can be verified.

Various related works have demonstrated the potential of schedules that employ layer fusion to reduce external memory accesses [7, 91, 165, 40]. However, the effect of these reductions on the net energy efficiency is typically overlooked. This chapter includes a high-level energy analysis based on the introduced schedule cost model, which results in some potentially surprising conclusions regarding the benefits of layer fusion.

The main contributions of this chapter are:

- Introduction of a platform agnostic, mathematical model of the cost of a CNN schedule in terms of memory accesses, memory footprint, and compute load, considering the vast scheduling space defined by loop interchange, loop tiling, loop fusion, recomputation, and explicit data transfer scheduling (section 5.3 & 5.4).
- An open source tool that implements the introduced model, enabling exhaustive design space exploration for CNNs [154] (section 5.5).
- Validation of the proposed models accompanied with detailed analysis of the effects of various scheduling techniques on six real-world networks (section 5.6).
- Generic energy evaluation using the modelled schedule costs, which shows that the reduction in external memory accesses achieved by layer fusion does not automatically translate to significant net energy reduction (section 5.7).

The remainder of this chapter is organised as follows. First, the related work on CNN scheduling is discussed in section 5.2. Next, the scheduling space is formally defined in section 5.3. The cost model defined on this space is introduced in section 5.4. Section 5.5 details the open source tool and experimental setup.

Results on model validation and design space exploration are provided and analysed in section 5.6. Section 5.7 contains an energy evaluation of the discovered schedules for a platform with a multi-level memory hierarchy. Finally, section 5.8 discusses current limitations and future work, and section 5.9 concludes the work.

5.2 Related Work

Deep neural networks are both compute and memory intensive algorithms, and only have become viable methods by the merit of increased compute capacity about a decade ago. The recent renewed interest in deep neural networks was initiated by the successes in image classification of general purpose computing on graphics processing units (GPGPU) based implementations of CNNs [25, 81] circa 2011-2012. Since then tremendous effort has been made to enable the efficient execution of deep neural networks on energy-constrained (embedded) devices. Because the basic algorithm does not change much over different applications, CNNs in essence provide a universal solution to many compute tasks. This makes them a highly eligible target for dedicated hardware solutions, and as such has inspired the design of many CNN hardware accelerators [127].

Because CNNs have a large memory footprint, these accelerators typically require a form of external memory to store the network parameters and intermediate results. One of the first published accelerator designs to recognize the importance of minimizing accesses to this external memory was the Eyeriss by Chen et al. [22]. Based on manual analysis an execution schedule of CNNs is proposed for Eyeriss, which exploits spatial features of the architecture, and leverages strip-mining (a subset of tiling) to handle networks that do not match naturally with the dimensioning of the compute elements inside the accelerator. The iteration order over the CNN operations is fixed however, and tailored towards the architecture.

Rather than fixing the iteration order, SmartShuttle [92] takes more of the scheduling space into account by defining three iteration strategies, each targeting capturing reuse of different data. In particular, there are schedules that primarily optimise capturing parameter/weight reuse, input feature map data reuse, or output feature map reuse. Selecting between these schedules depends on the dimensions of the network layers. This choice already starts to outline the trilemma of selecting of which data elements to capture the reuse in local buffers, and the difficulty of finding the schedule that minimises the overall external

memory accesses, as optimising for one subset of the data typically hurts the captured reuse of other parts.

To address the selection of a schedule based on tiling Peemen et al. [114] introduce generic formulas to enable fast schedule space exploration. The proposed formulas require manual tuning for different loop orders however, which is addressed by the model proposed by Waeijen et al. [160]. This model can generically compute the cost of schedules that include loop reordering, loop tiling, and explicit scheduling of data transfers. A very extensive framework that combines the same scheduling techniques with multi-level memory mapping is ZigZig [102]. Missing in these models and framework, however, is the capability to handle loop fusion, or layer fusion as first introduced by Alwani et al. [7].

Alwani et al. [7] introduce the concept of fusing the computation of two consecutive layers in order to avoid the transfer of intermediate results to external memory. In essence this is an on-demand computation of the intermediate results, which are immediately consumed by the next layer. Because of overlap in tiles, as the authors note, there is the option to recompute intermediate results of which not all uses fall within a single tile, or to store this subset of intermediate results. Eventually only schedules without recompute are considered however. The layer fusion proposed by Alwani et al. [7] suffers from another shortcoming, aptly dubbed the *computation pyramid* by the authors. This term refers to the phenomenon that when more layers are fused, the number of points a tile in the output layer depends on expands rapidly, creating a pyramid of dependencies towards the input of the network. The AivoTTA accelerator [68] exhibits this very same imperfection.

This particular issue is addressed by Li et al. [91] by modifying the CNN algorithm and removing several dependencies between layers, while maintaining acceptable accuracy. An arguably preferable solution which does not require changes to the CNN algorithm is proposed by Goetschalckx et al. [40], who employ line buffers to fuse layers, or execute *depth-first* in their terminology. This solution is more attractive as the functionality of the network remains unchanged, while large gains can be achieved in particular for networks with large layer dimensions. However, the proposed approach prohibits tiling in the channel dimension, and requires to always store the entire weight set on-chip, which may be highly suboptimal for networks with relatively small layer dimensions, i.e., layers of which the memory footprint is not dominated by data, but by weights. Despite these shortcomings, the depth-first methodology of Goetschalckx et al. [40] sets

Table 5.1: Set of structural parameters of a convolution layer.

Parameter	Description
D_x	Width of output (Y) feature maps
D_y	Height of output (Y) feature maps
D_z	Number of output (Y) feature maps
D_i	Number of input (X) feature maps
D_m	Convolutional kernel width
D_n	Convolutional kernel height

the standard for schedules with layer fusion, and will be used as an important benchmark throughout this chapter.

Finally, several recent works have included layer fusion in graphics processing unit (GPU) code generation [137, 181]. These works clearly show the potential gains of layer fusion, although both rely on heuristics to find good schedules, and require profiling runs on the target hardware. The models proposed in this chapter can be adapted towards single instruction multiple data (SIMD) and GPU execution as is further discussed in section 5.8, and can give insight into the performance of a schedule without execution on a target machine. Consequently they may be used to speedup and expand the design space searches of such heuristic (GPU) auto-schedulers.

5.3 Scheduling Space

To facilitate the definition of the cost models later in section 5.4, first the covered scheduling space is formally defined in this section. This definition starts with a high-level description of CNNs, followed by detailed descriptions of the considered scheduling techniques.

In a nutshell, a convolutional neural network functionally consists of a series of parallel, convolutional filters, or *layers*, connected by non-linear activation functions. The weights of these filters are determined during a learning phase in such a way that the network can perform its intended classification task. Once a suitable set of weights has been established, it remains static throughout the classification, or inference, phase, which is the focus of this chapter. Readers left desiring a more detailed description can find an excellent in-depth introduction to convolutional neural networks in the 'Deep Learning' book by Goodfellow et al. [41].

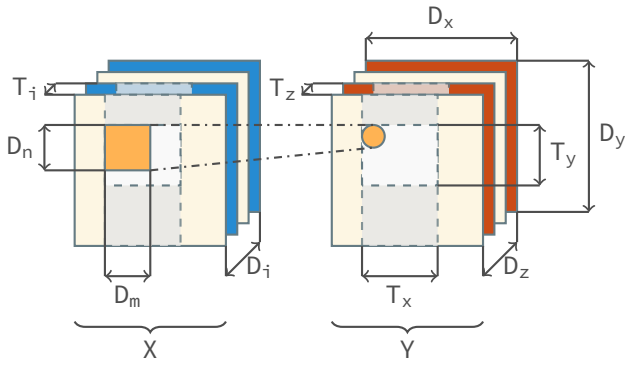


Figure 5.1: Single convolutional layer with input array X of and output array Y. Dimensional notation shown here is used throughout this chapter, where D_n and D_m denote the kernel height and width, D_i and D_z the number of feature maps in the input and output array, and finally D_x and D_y the width and height of the output array, respectively. Furthermore, T_i, T_z, T_y and T_x denote a tile size in the input feature maps, output feature maps, and output array height and width, respectively.

A single convolution layer consists of a set of convolution filters which are to be applied to a set of input surfaces to produce a set of output surfaces. These surfaces are referred to as *feature maps*. In the general case, a filter is applied for each pair of input and output feature maps, which is also the type of convolutional layer considered in the remainder of this chapter. More advanced layer types, such as depthwise convolution, are not directly considered, although possible model extensions are discussed in section 5.8.3.

Structurally a standard convolution layer is completely defined by the set of parameters listed in table 5.1. Figure 5.1 is a visual representation of such a layer, and its various dimensions. Convolutions are applied to the source feature maps on the left (X), and their results, after application of a non-linear transformation, are aggregated in the feature maps on the right (Y). From an implementation viewpoint, a convolutional layer is a deep loop nest. The pseudocode of a single layer is shown in code 5.1. Here variables Sx and Sy represent the stride of the filter on the input, which typically is one. A complete neural network consists of several of these layers connected through their feature maps. As such, a neural network can can represented as a directional graph $G(V, E)$ with the network layers V as nodes, and directional edges E to indicate their producer — consumer relationships.

```
1  for(int z=0; z<Dz; z++)
2   for(int y=0; y<Dy; y++)
3    for(int x=0; x<Dx; x++){
4     Y[z][y][x]=bias[z];
5     for(int i=0; i<Di; i++)
6      for(int n=0; n<Dn; n++)
7       for(int m=0; m<Dm; m++)
8        Y[z][y][x]+= \
9         X[i][y*Sy+n][x*Sx+m] \
10        * W[z][i][n][m];
11    Y[z][y][x]=act(Y[z][y][x]);
12   }
```

Code 5.1: Loop nest for a single convolution layer.

From code 5.1 it can be seen that the MAC operations (lines $8-10$) within a layer are completely independent. As such they may be executed in any order, yielding $(\mathsf{D_z} \times \mathsf{D_i} \times \mathsf{D_y} \times \mathsf{D_x} \times \mathsf{D_m} \times \mathsf{D_n})!$ scheduling options, ignoring the bias initialisation and the application of the activation function which even further increase the scheduling space. Reordering these operations will result in different reuse-distance distributions for the input data elements `X`, output data elements `Y`, and weights `W`. A smart reordering will capture more data-reuse in an internal buffer of given size, and as such minimize the accesses to an external memory. However, many of these schedules are highly irregular, and impossible to capture within reasonable code size. Therefore, this chapter only considers those schedules that can be generated using loop reordering and loop tiling at the layer level, as will be further discussed in section 5.3.1, section 5.3.2 respectively.

Apart from scheduling the compute operations, the data transfers between memory levels can also be explicitly scheduled. For accelerators that typically use scratchpad memory such scheduling is imperative, but machines using caches can also benefit from grouping data transfers. Explicitly scheduling these transfers consists of specifying *what* data will be stored and reused, and *when* this data is loaded from the external memory. To specify this, the store level and compute level concepts of the Halide language [124] are used, and made part of the considered scheduling space as described in section 5.3.3. Apart from scheduling the data transfers these concepts also allow for a precise expression of *recomputation* of intermediate results; a scheduling technique which provides a trade-off between (external) memory accesses and compute workload, as described in section 5.3.5.

Finally the scheduling space is expanded beyond scheduling individual layers by allowing layer fusion. Assume two convolutional layers A and B, which are connected in a network in such a way that B consumes the output of A. Following the dependencies, it is clear that some operations in B can already be executed, even if not all operations that belong to A are completed. Therefore it is possible to move (part of) the production of layer A into the loop nest of layer B using loop fusion. In this manner the results of layer A can potentially be consumed and discarded by B shortly after their production, effectively reducing their lifetime. Compared to an approach without layer fusion this has the potential to significantly reduce the accesses to a large external memory. The technique can furthermore be applied recursively, allowing any number of consecutive convolutional layers to be fused. Details on how this affects the overall scheduling space are provided in section 5.3.4.

5.3.1 Loop Reordering

To formally define the loop order of a schedule, let $\mathtt{L}$ denote the set of all loop variables in a convolutional layer. In accordance to code 5.1, $\mathtt{L} = \{\mathtt{z}, \mathtt{y}, \mathtt{x}, \mathtt{i}, \mathtt{m}, \mathtt{n}\}$. The loop order $O \subseteq \mathtt{L} \times \mathtt{L}$ defines a set of binary relations over $\mathtt{L}$, where, with $\mathtt{l}, \mathtt{l'} \in \mathtt{L}$, $\mathtt{l} \prec \mathtt{l'}$ yields true iff $\mathtt{l}$ is inner to $\mathtt{l'}$ in the loop nest, resulting in a total ordering of $\mathtt{L}$. In code 5.1, the following expression holds: $\mathtt{m} \prec \mathtt{n} \prec \mathtt{i} \prec \mathtt{x} \prec \mathtt{y} \prec \mathtt{z}$.

Loop order $O \subseteq \mathtt{L} \times \mathtt{L}$ always results in a total ordering. To indicate a position in this ordering, we define the term *loop level*, which is independent of a particular ordering. E.g, in code 5.1 loop `m` has taken the inner most loop level. The inner most level indicates the inner most loop, independent of which loop (variable) is assigned to this inner most position by the loop order.

With this definition of loop order in place, consider the reuse distance of data elements in the `X` and `Y` arrays. Note that the accesses to array `Y` are independent of loop variable `i` (line 8 in code 5.1), while those to array `X` (line 9 in code 5.1) are dependant on `i`. Henceforth, because $\mathtt{i} \prec \mathtt{z}$ in code 5.1, the accumulations to a single `z`-coordinate in the `Y` array on line 8 are relatively close in time. However, for each of these accumulations an element from a unique `i` index has to be loaded from the `X` array on line 9. Therefore the reuse distances on array `X` are relatively long, while those on array `Y` are relatively short. Yet, when loops `i` and `z` are interchanged, i.e., $\mathtt{z} \prec \mathtt{i}$, the reverse holds. Which of these orders is favourable depends, amongst other factors, on the particular dimensioning of the layer. To complicate matters further, the other loops can also be reordered,

and the data reuse of array $\mathtt{W}$ is also significant, rendering a complex trade-off. Nonetheless, it can be stated that moving kernel loops $\mathtt{m}$ and $\mathtt{n}$ will likely not be beneficial, as typically $\mathtt{D_m}$ and $\mathtt{D_n}$ are very small (common values encountered in practice include one, three, and five). As such, loop reordering in this chapter is restricted to the remaining loop levels.

5.3.2 Loop Tiling

Tiling is a classic scheduling technique to alter the execution order of operations. As discussed in the previous section, a particular loop order may decrease the reuse on one data array, but increase it in another one. A different loop order may achieve the reverse. Loop tiling enables a hybrid approach, allowing a balanced average reuse distance for all data accesses. By splitting a loop $\mathtt{l} \in \mathtt{L}$ that iterates over a complete dimensions into an inner part $\mathtt{li}$, and outer part $\mathtt{lo}$, it is possible to only compute part of a dimension inner to the iteration over another dimension.

For CNN layers in particular, each loop in code 5.1 can be split. Again, because the kernel dimension $\mathtt{D_m}$ and $\mathtt{D_n}$ are typically very small, tiling loops $\mathtt{m}$ and $\mathtt{n}$ are not considered. However, the remaining loops, i.e., $\{\mathtt{i}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$, can all be tiled into parts of size $\mathtt{T_l}$, where $\mathtt{l} \in \{\mathtt{i}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$. Since $\mathtt{T_l}$ can be set to one, it is possible to rewrite code 5.1 into code 5.2 without loss of generality.

For the remainder of this chapter, code 5.2 will be used to define schedules of a single layer. As such, a tiled schedule formally consists of an ordering O on the set of tiled loop variables $\mathtt{TL} = \{\mathtt{zo}, \mathtt{yo}, \mathtt{xo}, \mathtt{io}, \mathtt{zi}, \mathtt{yi}, \mathtt{xi}, \mathtt{ii}, \mathtt{n}, \mathtt{m}\}$, and a set of tile sizes $\mathtt{T} = \{\mathtt{T_z}, \mathtt{T_y}, \mathtt{T_x}, \mathtt{T_i}\}$.

5.3.3 Store & Compute Levels

Besides the computations, transfers between external memory and local buffers can be scheduled explicitly as well. To capture these memory operations the *store and compute level* concepts from the Halide language [124] are employed. These levels are defined for each array $\mathtt{X}$, $\mathtt{Y}$, and $\mathtt{W}$, and dictate respectively *what* data volume is transferred *when*.

```
1  //outer tile loops
2  for(int zo=0; zo<Dz; zo+=Tz)
3   for(int yo=0; yo<Dy; yo+=Ty)
4    for(int xo=0; xo<Dx; xo+=Tx)
5     for(int io=0; io<Di; io+=Ti)
6      //inner tile loops
7      for(int zi=zo; zi<zo+Tz; zi++)
8       for(int yi=yo; yi<yo+Ty; yi++)
9        for(int xi=xo; xi<xo+Tx; xi++){
10         if(io==0)
11          Y[zi][yi][xi]=bias[zi];
12         for(int ii=io; ii<io+Ti; ii++)
13          for(int n=0; l<Dn; n++)
14           for(int m=0; m<Dm; m++)
15            Y[zi][yi][xi]+= \
16               X[ii][yi*Sy+n][xi*Sx+m] \
17               * W[ii][zi][m][n];
18         if(io+Ti>=Di)
19          Y[zi][yi][xi]= \
20           act(Y[zi][yi][xi]);
21        }
```

Code 5.2: Tiled loop nest for a single convolution layer.

Let $\mathtt{ARR} = \{\mathtt{X}, \mathtt{Y}, \mathtt{W}\}$ denote the set of all data arrays in a layer. The store level is then defined as follows:

> The **store level** $SL_{\mathtt{arr}} \in \mathtt{TL}$ for array $\mathtt{arr} \in \mathtt{ARR}$ determines that at *all* data accesses to elements in $\mathtt{X}$ inside a single iteration of loop $SL_{\mathtt{arr}}$ have to be served from local memory after an initial load.

In code 5.2, for example, if $SL_{\mathtt{X}}$ is set to the loop level assigned to `xi` on line 9, the data required for the $(\mathtt{T_i} \times \mathtt{D_n} \times \mathtt{D_m})$ operations inside one iteration of `xi` need to be served from local memory. Note that there is also data reuse of elements in $\mathtt{X}$ between two iterations of loop `xi`, as illustrated in figure 5.2. In particular, there is an overlap of $(\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{D_m} - 1))$ elements between two consecutive iterations[1]. The store level does not specify to capture this reuse in a local memory. To capture this reuse, the store level has to be moved one loop level up, to `yi`. Since one iteration of loop `yi` encapsulates $\mathtt{T_x}$ iterations of loop `xi`, the reuse between these iterations must now be captured by the internal buffer as well. As a consequence the volume of data that needs to be captured increases from $(\mathtt{T_i} \times \mathtt{D_m} \times \mathtt{D_n})$ to $(\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{T_x} + \mathtt{D_m} - 1))$, as visualised in figure 5.2.

[1] Assuming stride $\mathtt{Sx} = 1$ for simplicity

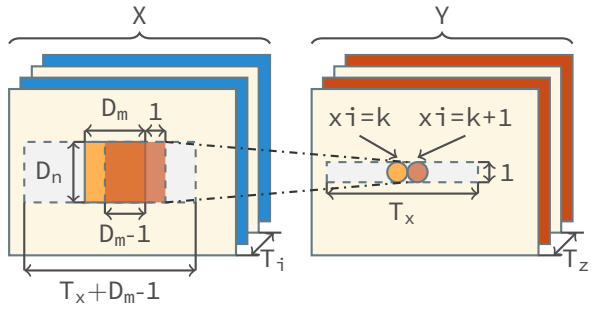


Figure 5.2: Overlap of $(\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{D_m} - 1))$ elements in the input data `X` between iterations '`n`' and '`n+1`' of loop `xi` in code 5.2. N.B. The volume required for $SL_{\mathtt{X}} = \mathtt{yi}$ is $(\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{T_x} + \mathtt{D_m} - 1))$, yet a rotating buffer of size $(\mathtt{T_i} \times \mathtt{D_n} \times \mathtt{D_m})$ is sufficient.

Note that $\mathtt{T_x}$ is defined on the output layer, and because of the kernel size $\mathtt{D_m}$ a tile of size $(\mathtt{T_x} + \mathtt{D_m} - 1)$ is thus required of the input layer. This demonstrates the trade-off between required on-chip buffer size and number of external memory accesses that can be explored using the store level.

Apart from the *store* level $SL_{\mathtt{arr}} \in \mathtt{TL}$, also a *compute* level is defined:

> The **compute level** $CL_{\mathtt{arr}} \in \mathtt{TL}$ determines at what loop iteration new data is produced/loaded for each array $\mathtt{arr} \in \mathtt{ARR}$.

This additional directive enables an optimization known as buffer *folding*. For $SL_{\mathtt{X}} = \mathtt{yi}$ the data volume that has to be delivered by on-chip memory is equal to $(\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{T_x} + \mathtt{D_m} - 1))$ elements[1], but that does not require that this data is all live at the same time. In fact, as the $(\mathtt{T_i} \times \mathtt{D_m} \times \mathtt{D_n})$ kernel moves from left to right as the `xi` loop proceeds, old data to the left will no longer be reused within the current iteration of loop `yi`. By selecting `xi` as the compute level of array `X`, i.e., $CL_{\mathtt{X}} = \mathtt{xi}$, new data is only produced, i.e., fetched from external memory, at each iteration of `xi`. Since there are $(\mathtt{T_i} \times \mathtt{D_n} \times (\mathtt{D_m} - 1))$ elements overlap between each iteration, as discussed before, for each iteration only $\mathtt{T_i} \times \mathtt{D_n}$ new elements are required. These can be kept in a rotating buffer of only $(\mathtt{T_i} \times \mathtt{D_m} \times \mathtt{D_n})$ elements. Note that the reuse captured by the internal buffer is unchanged, and dictated only by the store level. The addition of the compute level enables folding of buffers, such that the same reuse can be captured with less buffer space. Combined, the store and compute levels respectively dictate *what* data is transferred *when*.

Unlike loop ordering and tiling, the store and compute levels can not be chosen freely. In particular, data dependencies dictate that the production of the weights and input must be scheduled before, or in parallel with, the production of the feature maps, i.e., $SL_{\mathsf{Y}} \preceq SL_{\mathsf{W}}$ and $SL_{\mathsf{Y}} \preceq SL_{\mathsf{X}}$. Furthermore, the store level is always to be selected from one of the inner loops, or one level higher, i.e., any loop in code 5.2 between lines 5–12. Setting the store level any higher would encompass at least one outer and inner loop of the same dimension, cancelling the effect of tiling. The same can be achieved by equating the tile size to the dimension, and as such these schedules are covered without the need to consider the remaining outer loop levels.

Finally the compute level of a data array should always be equal to, or lower than the store level of that array, i.e., $CL_{\mathsf{arr}} \preceq SL_{\mathsf{arr}}$. This requirement originates from the trivial dependency between the production of an element and the allocation of its storage. If no storage is allocated, the element can not be produced.

5.3.4 Layer Fusion

Apart from reordering computations within a layer, as performed by loop reordering and tiling, there is also the possibility to reorder operations between layers. In particular, if one layer is computed partially, some of the computations of the succeeding layer may already have all their input operands ready, enabling their execution. This concept is best described in terms of producers and consumers, where a first layer produces data which is consumed by a second layer. Rather than computing the producer completely before starting the computation of the consumer, the computation of the producer can be inlined to the computation of the consumer. Again, these transformations alter reuse distances and lifetimes of the various data arrays. Critically, the results from the producer can be consumed much earlier. Unless there is already sufficient on-chip memory to buffer an entire layer, loop fusion can be used to consume the results of intermediate layers, rather then sending them out to external memory only to be retrieved again later.

The data dependencies between two convolutional layers are illustrated in figure 5.3. Note that for a layer v fused into a layer u, the output array Y of layer v is the same as the input array X of u. Generically, $\mathsf{Y}_k = \mathsf{X}_{k+1}$ Therefore, in figure 5.3, the Y arrays have been named by their X array equivalent. In this figure a tile of size $\mathsf{T_z} \times \mathsf{T_x} \times \mathsf{T_y}$ is to be produced in array `X2`. Assume `X1` is not yet computed. When the production of $\mathsf{T_z} \times \mathsf{T_x} \times \mathsf{T_y}$ is about to start, first

a tile in X1 of size $\mathtt{T_i} \times (\mathtt{T_x} + \mathtt{D_{m1}} - 1) \times (\mathtt{T_y} + \mathtt{D_{n1}} - 1)$ is produced. Once this tile is ready, the computation of $\mathtt{T_z} \times \mathtt{T_x} \times \mathtt{T_y}$ in X1 commences.

The basic code of two fused layers is given in code 5.3. As can be seen, the production of a $\mathtt{T_i} \times (\mathtt{T_x} + \mathtt{D_{m1}} - 1) \times (\mathtt{T_y} + \mathtt{D_{n1}} - 1)$ sized tile X1 is inlined in the loop nest of X2. This technique is generically known as loop fusion. Since in this particular context it is applied to loop nests of CNN layers the term *layer fusion* is used.

Although not shown for simplicity in code 5.3, it is entirely possible to also tile and reorder the production of the inlined producer. From this perspective, tiling and reordering are orthogonal concepts to layer fusion. Furthermore, layer fusion can be applied recursively, fusing an unlimited number of consecutive layers. This increases the scheduling space tremendously, complicating the task of finding an optimal schedule for a given network.

The connections between layers in a neural network can generically be captured in a directed graph $G(V, E)$ where V represents the set of individual layers and their associated structural parameters as listed in table 5.1, and E is a set of tuples (src, dst) with src, $dst \in V$ that define a directional relation from src to dst. The production of a layer may be fused into one of its direct successors in this network graph G or it may not be fused at all. To denote this, each layer $v \in V$ is assigned a fuse target $Fuse(v) \in successors(v) \cup \{v\}$, where $successors(v) = \{v' \mid (v, v') \in E\}$ is the set of all direct successors of layer v in $G(V, E)$. The production of layer v is then scheduled inline into the production of $Fuse(v)$. When the fuse target is set to layer v itself, the layer is consequently not fused.

5.3.5 Recomputation

Apart from shortened data lifetimes, layer fusion also introduces another interesting trade-off. As discussed, depending on tiling and store levels not all data reuse may be captured from a local buffer. The same naturally holds for the data of an intermediate, fused layer. In figure 5.3 the data of X1 has multiple uses in the production of X2. If not all uses of an element are captured, there is the option to store the intermediate value of X1 in external memory and reload it for future uses. Alternatively it can be discarded, and *recomputed* from X0 when it is needed again. In this way a trade-off can be made between compute load and external memory traffic. This is particularly interesting for modern and

```
1  //outer tile loops of X1->X2
2  for(int zo=0; zo<Dz; zo+=Tz)
3   for(int yo=0; yo<Dy; yo+=Ty)
4    for(int xo=0; xo<Dx; xo+=Tx)
5     for(int io=0; io<Di; io+=Ti){
6      //Inline production of X0->X1
7      for(int z=0; z<Ti; z++)
8       for(int y=0; y<Ty+Dn1-1; y++)
9        for(int x=0; x<Tx+Dm1-1; x++){
10        X1[z][y][x]=bias[z];
11        for(int i=0; i<Di0; i++)
12         for(int n=0; n<Dn0; n++)
13          for(int m=0; k<Dm0; m++)
14           X1[z][y][x]+= \
15            X0[i][y*Sy0+n][x*Sx0+m] \
16            * W01[z][i][n][m];
17        X1[z][y][x]=act(X1[z][y][x]);
18       }
19     //inner tile loops of X1->X2
20     for(int zi=zo; zi<zo+Tz; zi++)
21      for(int yi=yo; yi<yo+Ty; yi++)
22       for(int xi=xo; xi<xo+Tx; xi++){
23        if(io==0)
24         X2[zi][yi][xi]=bias[zi];
25        for(int ii=io; ii<io+Ti; ii++)
26         for(int n=0; n<Dn1; n++)
27          for(int m=0; k<Dm1; m++)
28           X2[zi][yi][xi]+= \
29            X1[ii][yi*Sy1+n][xi*Sx1+m]\
30            * W12[ii][zi][m][n];
31        if(io+Ti>=Di)
32         X2[zi][yi][xi]= \
33          act(X2[zi][yi][xi]);
34       }
35    }
```

Code 5.3: Code for 2 fused layers as illustrated in figure 5.3.

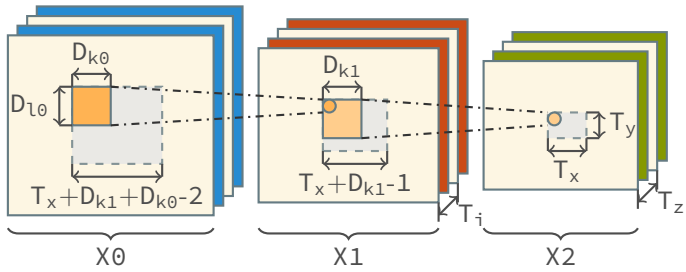


Figure 5.3: Three state arrays X of two consecutive convolutional layers. To produce tile $\mathtt{T_z} \times \mathtt{T_x} \times \mathtt{T_y}$ on X2, a tile of $\mathtt{T_i} \times (\mathtt{T_x} + \mathtt{D_m} - 1) \times (\mathtt{T_y} + \mathtt{D_n} - 1)$ is required from array X1. In a fused schedule, this tile of X1 is produced in-line to the production of the tile in X2, rather than first computing X1 completely.

Table 5.2: Layer schedule s of a convolution layer $v \in V$.

Parameter	Description
$O \subseteq \mathtt{TL} \times \mathtt{TL}$ *	Loop ordering
$\mathtt{T_z}, \mathtt{T_y}, \mathtt{T_x}, \mathtt{T_i} \leq \mathtt{D_z}, \mathtt{D_y}, \mathtt{D_x}, \mathtt{D_i}$	Tile sizing
$SL_\mathtt{X}$, $SL_\mathtt{Y}$, $SL_\mathtt{W} \in \mathtt{TL}$	Store levels
$CL_\mathtt{X}$, $CL_\mathtt{Y}$, $CL_\mathtt{W} \in \mathtt{TL}$	Compute levels
$Fuse \in successors(v) \cup \{v\}$ **	Fuse target

* Recall, $\mathtt{TL} = \{\mathtt{zo}, \mathtt{zi}, \mathtt{yo}, \mathtt{yi}, \mathtt{xo}, \mathtt{xi}, \mathtt{io}, \mathtt{ii}\}$ denotes the set of all tiled loop levels in a convolution layer.
** Note, when $Fuse = v$ the layer is not fused.

future technology nodes, where (re)compute typically can be orders of magnitude cheaper in both time and energy than re-accessing external memory [124].

5.3.6 Formal Schedule

As stated, a convolutional layer is structurally defined by the set of parameters listed in table 5.1. For each layer $v \in V$, where V represents the complete set of layers that make up a particular CNN, a layer schedule s can be defined according to the various scheduling options discussed in this section. Such a scheduled layer s consists of the parameters listed in table 5.2. A network schedule $S = \{(v, s) \mid v \in V\}$ is consequently defined as the set of tuples of layers and accompanying layer schedules for each layer in the network.

Table 5.3: Model Summary.

Function	Description
$\text{Buf}_\mathtt{W}$, $\text{Buf}_\mathtt{X}$, $\text{Buf}_\mathtt{Y}$	Buffer sizes of the weight, input, and output arrays respectively.
$\text{Acc}_\mathtt{W}$, $\text{Acc}_\mathtt{X}$, $\text{Acc}_\mathtt{Y}$	Number of weight, input and output elements respectively transferred from/to external memory.
MACS	Total number of MACs.

5.4 Cost Models

For real-world neural networks, merely iterating through the entire scheduling space as described in section 5.3 already presents a significant task. Benchmarking each of these schedules on a target machine to find the best match is simply intractable. This section describes a set of mathematical expressions which, given a network schedule, accurately model the required number of *external memory accesses*, the required *internal buffer space*, and the number of computations measured in MACs, as summarized in table 5.3. These expressions only require a handful of computations compared to benchmarking a network schedule on a target machine, and as such enable fast design space exploration. The remainder of this section defines these expressions precisely. Readers primarily interested in applying these models may skip ahead to section 5.5 which introduces the open source implementation of these equations in the form of the `ConvFuser` tool [154]. Also the final results in section 5.6 can be interpreted without in-depth understanding of the detailed model presented in this section.

5.4.1 Prerequisites

To aid the formulation of these models, a number of notational shorthands and auxiliary functions are defined first. In general, the multiplication of each element in an arbitrary set S will be abbreviated to $\prod S$, i.e.,

$$\prod S = \prod_{s \in S} s.$$

Note that in accordance with the common definition of the product operator, the product of the empty set $\emptyset$ is defined as one.

Given a layer and an associated schedule $(v, s) \in S$, the (sub)set of structural dimensions of layer $v \in V$ as defined in table 5.1, and the (sub)set of tile sizes in

schedule s as defined in table 5.2, that belong to a given (sub)set of loop levels $\mathtt{L'} \subseteq \mathtt{L}$, is defined by the following two auxiliary functions respectively:

$$D\left(v, \mathtt{L'}\right) = \left\{\mathtt{D_l} \mid \mathtt{l} \in \mathtt{L'} \wedge \mathtt{D_l} \in v\right\},$$

$$T\left(s, \mathtt{L'}\right) = \left\{\mathtt{T_l} \mid \mathtt{l} \in \mathtt{L'} \wedge \mathtt{T_l} \in s\right\}.$$

Furthermore a translation function $\kappa\left(v, \mathtt{l}\right)$ is defined, with layer $v \in V$ and loop level $\mathtt{l} \in \{\mathtt{x}, \mathtt{y}\}$. This function converts loop levels $\mathtt{x}$ and $\mathtt{y}$ to their spatially related kernel dimensions $\mathtt{D_m}$ and $\mathtt{D_n}$ respectively:

$$\kappa\left(v, \mathtt{l}\right) = \begin{cases} \mathtt{D_m} \in v & \mathtt{l} = \mathtt{x} \\ \mathtt{D_n} \in v & \mathtt{l} = \mathtt{y} \end{cases}.$$

The corresponding set operator K, which translates all loop levels in a set $\mathtt{L'} \subseteq \mathtt{L}$, is defined as:

$$K\left(v, \mathtt{L'}\right) = \left\{\kappa\left(v, \mathtt{l}\right) \mid \mathtt{l} \in \mathtt{L'}\right\}.$$

Another helper function translates a loop level $\mathtt{l} \in \mathtt{L}$ into the corresponding inner tiled loop level $\mathtt{li} \in \mathtt{TL}$:

$$inner\left(\mathtt{l}\right) = \mathtt{li}.$$

Since for many models it matters whether or not the inner loop of a particular loop level $\mathtt{l} \in \mathtt{L}$ is preceded by the store level in a given layer schedule s, the set of all loop levels in set $\mathtt{L'} \subseteq \mathtt{L}$ which are preceded by the store level $SL_{\mathtt{arr}}$ of array $\mathtt{arr} \in \mathtt{ARR}$, i.e., the collection of loop levels below/inner to the store level for array $\mathtt{arr}$, is defined as:

$$LT_{\mathtt{arr}}\left(s, \mathtt{L'}\right) = \left\{\mathtt{l} \mid \mathtt{l} \in \mathtt{L'} \wedge inner\left(\mathtt{l}\right) \prec SL_{\mathtt{arr}}\right\},$$

where $SL_{\mathtt{arr}} \in \mathtt{TL} \in s$, $\prec \in O$, and $O \in s$.

The complement of this set, i.e., the set of loop levels which are equal to or above/outer to the store level, is defined as:

$$GE_{\mathtt{arr}}\left(s, \mathtt{L'}\right) = \mathtt{L'} - LT_{\mathtt{arr}}\left(s, \mathtt{L'}\right).$$

Furthermore the set of folded loop levels F, i.e., the levels between the store and compute level is defined as:

$$F_{\mathtt{arr}}\left(s, \mathtt{L'}\right) = \left\{\mathtt{l} \mid \mathtt{l} \in \mathtt{L'} \wedge CL_{\mathtt{arr}} \preceq inner(\mathtt{l}) \prec SL_{\mathtt{arr}}\right\},$$

Table 5.4: Helper Functions Summary.

Function	Description
$D(v, \texttt{L'})$	Set of loop dimensions $\texttt{D}_\texttt{l}$ belonging to loop levels $\texttt{l} \in \texttt{L'} \subseteq \texttt{L}$
$T(s, \texttt{L'})$	Set of tile dimensions $\texttt{T}_\texttt{l}$ belonging to loop levels $\texttt{l} \in \texttt{L'} \subseteq \texttt{L}$ in schedule s
$\kappa(v, \texttt{l})$	Translation of loop level $\texttt{l} \in \{\texttt{x}, \texttt{y}\}$ to loop dimension of corresponding kernel in layer v
$K(v, \texttt{L'})$	Application of $\kappa(v, \texttt{l})$ to complete set of loop levels $\texttt{l} \in \texttt{L'} \subseteq \texttt{L}$
$inner(\texttt{l}) = \texttt{li}$	Translates a non-tiled loop level $\texttt{l} \in \texttt{L}$ into the corresponding inner tiled loop level $\texttt{li} \in \texttt{TL}$
$LT_{\texttt{arr}}(s, \texttt{L'})$	Set of loop levels in $\texttt{L'} \subseteq \texttt{L}$ that are below or equal to the store level in layer schedule s
$GE_{\texttt{arr}}(s, \texttt{L'})$	Set of loop levels in $\texttt{L'} \subseteq \texttt{L}$ that are above the store level in layer schedule s
$F_{\texttt{arr}}(s, \texttt{L'})$	Set of loop levels in $\texttt{L'} \subseteq \texttt{L}$ that are in between the compute and store level in layer schedule s
$FuseSel(v, s, A, B)$	If layer v is fused in layer schedule s return set A, else return set B.

where $SL_{\texttt{arr}}, CL_{\texttt{arr}} \in \texttt{TL} \in s$, $\prec \in O$, $O \in s$, and the operator $(\texttt{l} \preceq \texttt{l'}) = (\texttt{l} \prec \texttt{l'} \vee \texttt{l} = \texttt{l'})$. Finally a set selection function is defined, which selects set A if the layer is fused, or set B otherwise.

$$FuseSel(v, s, A, B) = \begin{cases} A & fuse \neq v \\ B & \text{o.w.} \end{cases},$$

with fuse target $fuse \in s$.

For all these helper functions, when it is clear only a single layer v or schedule s is described, the v and s arguments are omitted for further brevity. An overview of these helper functions is provided in table 5.4.

5.4.2 Internal Memory Footprint

With this notation in place, the required internal buffer size of a single scheduled layer $(v, s) \in S$ can be concisely and accurately modelled. This buffer size is comprised of three parts, the sum of the memory footprints of the X, Y, and W arrays respectively. All these footprints can be obtained by computing the

Table 5.5: Memory footprint contributions of W for all loop levels.

Loop Lvl (l)	$SL_{\mathtt{W}} \prec \mathtt{l}$	$SL_{\mathtt{W}} \succeq \mathtt{l}$
x	1	1
y	1	1
m	$\mathtt{D_m}$	—
n	$\mathtt{D_n}$	—
z	$\mathtt{T_z}$	1
i	$\mathtt{T_i}$	1

volume of data below the respective store level $SL_{\mathtt{arr}}$, since this is the volume that will be loaded by the scheduled external memory access.

In general, for each loop level $\mathtt{l} \in \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{i}, \mathtt{m}, \mathtt{n}\}$ a selection has to be made between two options for each data array $\mathtt{arr} \in \{\mathtt{X}, \mathtt{Y}, \mathtt{W}\}$, one contribution to the data volume if said dimension is below $SL_{\mathtt{arr}}$, and one when it is equal or above. The product of these contributions yields the complete data volume.

Weight Array

For weight array W these options for the dimensions are explicitly listed in table 5.5. Since the accesses to W are independent of loop levels x and y, as can be seen in section 5.3 of code 5.1, these loop levels do not contribute to the memory footprint of W (set to 1 for unit operation in the final product). For loop levels m and n the full dimension $\mathtt{D_m}$ and $\mathtt{D_n}$ has to be counted respectively, since these loops are excluded from tiling in the defined schedule space, and they are also always below the store level. More interesting are loop levels z and i, which require a full tile $\mathtt{T_z}$ or $\mathtt{T_i}$ to be counted when they are below the store level, or only a single slice if they are not. The product of all correct contributions in table 5.5 yields the *initial* memory footprint of the W array. However, care has to be taken when the compute level $CL_{\mathtt{W}}$ is below the store level, and buffer folding is applied. In the case that z and/or i are folded (below the store level, but above or equal to the compute level), they only contribute as if they were above the store level.

Using the introduced notation, the memory footprint of the weight array W of a single convolution layer can be expressed as follows:

$$\begin{aligned}\mathrm{Buf}_{\mathtt{W}} = &\prod T(LT_{\mathtt{W}}(\Lambda) - F_{\mathtt{W}}(\Lambda)) \times \\ &\prod D(\{\mathtt{m}, \mathtt{n}\}) \times FuseSel(\mathtt{D_i}, 1),\end{aligned}$$

where $\Lambda = FuseSel(\{\mathtt{z}\}, \{\mathtt{z}, \mathtt{i}\})$. I.e., for the kernel loops $\mathtt{m}$ and $\mathtt{n}$ the full dimensions $\mathtt{D_m}$ and $\mathtt{D_n}$ are counted. For the non-fused case as described in table 5.5, the tile sizes of loop dimensions $\mathtt{z}$ and $\mathtt{i}$ are taken into account, provided they are below the store level and not folded. When the layer is fused into a successor however, the computation changes slightly because dimension $\mathtt{i}$ can no longer be tiled. Considering the nonlinear activation function on section 5.3 of code 5.1, all contributions in the $\mathtt{i}$ dimension have to be reduced before this activation can be applied, and the next layer can start its dependent computations. This can be more clearly seen in code 5.3, where, due to the activation function on section 5.3.4, the complete (untiled) $\mathtt{i}$ loop on section 5.3.4 has to be computed before the next layer can start production on section 5.3.4. Thus, $\mathtt{i}$ can not be tiled and instead the full $\mathtt{D_i}$ dimension is required, as is covered by the $FuseSel$ selection function.

Input Array

For input array $\mathtt{X}$ a similar equation can be derived. The notable differences are that $\mathtt{X}$ is independent of loop level $\mathtt{z}$, and that at loop levels $\mathtt{x}$ and $\mathtt{y}$ at least $\mathtt{D_m}$ and $\mathtt{D_n}$ input elements are required. When a tile is required in these dimensions, i.e. $\mathtt{x} \prec SL_{\mathtt{X}}$ or $\mathtt{y} \prec SL_{\mathtt{X}}$, the kernel size also comes into play and, as illustrated in figure 5.2, a contribution of $\mathtt{T_x} + \mathtt{D_m} - 1$ or $\mathtt{T_y} + \mathtt{D_n} - 1$ is required respectively. Another complicating factor is formed by the strides in the $\mathtt{x}$ and $\mathtt{y}$ dimensions, which change these terms to $(\mathtt{T_x} - 1) \times \mathtt{S_x} + \mathtt{D_m}$ and $(\mathtt{T_y} - 1) \times \mathtt{S_y} + \mathtt{D_n}$ respectively. The resulting memory footprint of a single layer is captured by the following expression:

$$\begin{aligned}\mathrm{Buf}_{\mathtt{X}} = & FuseSel\left(\mathtt{D_i}, \prod T\left(LT_{\mathtt{X}}\left(\{\mathtt{i}\}\right) - F_{\mathtt{X}}\left(\{\mathtt{i}\}\right)\right)\right) \times \\ & \prod K\left(GE_{\mathtt{X}}\left(\{\mathtt{x}, \mathtt{y}\}\right) \cup F_{\mathtt{X}}\left(\{\mathtt{x}, \mathtt{y}\}\right)\right) \times \\ & \prod \{(\mathtt{T_d} - 1) \times \mathtt{S_d} + \kappa(\mathtt{d}) \mid \\ & \quad \mathtt{d} \in LT_{\mathtt{X}}\left(\{\mathtt{x}, \mathtt{y}\}\right) - F_{\mathtt{X}}\left(\{\mathtt{x}, \mathtt{y}\}\right)\}.\end{aligned}$$

Again, $FuseSel$ is used to account for the full dimension in $\mathtt{i}$ when the layer is fused, for the same reason as in $\mathrm{Buf}_{\mathtt{W}}$, i.e., due to the nonlinear activation function on section 5.3 in code 5.1.

Output Array

Finally, the memory footprint of the output array Y is relatively straightforward, and depends on x, y, and z. For each of these dimensions the contribution is equal to the tile size, unless the dimension is above the store level or folded. That is, unless the current layer will be fused into the next layer, in which case the output array Y is effectively replaced by data array X of the next layer, yielding no memory contribution for Y. Furthermore it is important to note that once an output is complete, i.e., all $\mathtt{D_i}$ contributions of the preceding layer have been processed and the activation function is applied, there is no need to keep the completed output on-chip. From a memory viewpoint this resembles buffer folding, which is also how this optimization is taken into account in the final equation. The footprint contributions of those loop levels in {x, y, z} which are above i fold to one, resulting in:

$$\text{Buf}_{\mathtt{Y}} = FuseSel\left(0, \prod T(LT_{\mathtt{Y}}(\{\mathtt{x},\mathtt{y},\mathtt{z}\}) - F_{\mathtt{Y}}(\{\mathtt{x},\mathtt{y},\mathtt{z}\}))\right)$$

5.4.3 External Memory Accesses

The number of required external memory accesses can be derived in a similar manner as the internal memory footprint. The crucial difference is to not only account for the data volume transferred, but also how many times such a transfer takes place. These two terms can be considered separately, such that for data array $\mathtt{arr} \in \mathtt{ARR}$, the external memory accesses $\text{Acc}_{\mathtt{arr}}$ are expressed as the volume of a data transfer $\text{Vol}_{\mathtt{arr}}$, multiplied by the number of those transfers $\text{Trans}_{\mathtt{arr}}$:

$$\text{Acc}_{\mathtt{arr}} = \text{Vol}_{\mathtt{arr}} \times \text{Trans}_{\mathtt{arr}}.$$

Weight Array

For weight array W, the volume of a transfer $\text{Vol}_{\mathtt{W}}$ is nearly identical to its internal memory footprint $\text{Buf}_{\mathtt{W}}$, with the notable exception that for the transfer volume buffer folding has no effect. The fact that the buffer is smaller due to liveness of the variables does not invalidate the requirement to transfer the complete volume eventually. The transfer volume of array W can therefore be expressed as:

$$\begin{aligned}\text{Vol}_{\mathtt{W}} = &\prod T\left(LT_{\mathtt{W}}\left(\Lambda\right)\right) \times \\ &\prod D\left(\{\mathtt{m},\mathtt{n}\}\right) \times FuseSel(\mathtt{D_i}, 1),\end{aligned}$$

where $\Lambda = FuseSel(\{\mathsf{z}\}, \{\mathsf{z}, \mathsf{i}\})$. Note that when the current layer is fused into the next, all inputs $\mathsf{D_i}$ need to be handled before the results can be passed to the next layer. Again, the nonlinear activation function on section 5.3 of code 5.1 prevents partial updates of only $\mathsf{T_i}$ inputs to be consumed.

Next the number of transfers is to be determined. In general, if a loop level l is above or equal to the store level, the associated volume needs to be transferred for every $\mathsf{D_l}$ iterations. When l is beneath the store level however, that volume will have to be transferred only $\left\lceil \frac{\mathsf{D_l}}{\mathsf{T_l}} \right\rceil$ times. The ceiling operator is used here to arrive at a conservative bound, which accounts for a full tile transfer in case tile size $\mathsf{T_l}$ is not an exact multiple of $\mathsf{D_l}$. For weight array W the number of transfers is given by:

$$\mathrm{Trans}_{\mathsf{W}} = \prod D\left(GE_{\mathsf{W}}(\Lambda)\right) \times \prod \left\{ \left\lceil \frac{\mathsf{D_d}}{\mathsf{T_d}} \right\rceil \;\middle|\; \mathsf{d} \in LT_{\mathsf{W}}(\Lambda) \right\},$$

where $\Lambda = FuseSel(\{\mathsf{x}, \mathsf{y}, \mathsf{z}\}, \{\mathsf{x}, \mathsf{y}, \mathsf{z}, \mathsf{i}\})$ is used to compensate when due to fusion the entire volume is transferred.

Input Array

For data array X the transfer volume resembles the internal buffer size of array X, $\mathrm{Buf}_{\mathsf{X}}$, again ignoring any buffer folding:

$$\begin{aligned}\mathrm{Vol}_{\mathsf{X}} = & FuseSel\left(\mathsf{D_i}, \prod T\left(LT_{\mathsf{X}}(\{\mathsf{i}\})\right)\right) \times \\ & \prod K\left(GE_{\mathsf{X}}(\{\mathsf{x}, \mathsf{y}\})\right) \times \\ & \prod \left\{(\mathsf{T_d} - 1) \times \mathsf{S_d} + \kappa(\mathsf{d}) \mid \mathsf{d} \in LT_{\mathsf{X}}(\{\mathsf{x}, \mathsf{y}\})\right\}.\end{aligned}$$

The number of transfers for X is in fact equal to those of W ($\mathrm{Trans}_{\mathsf{W}}$), apart from checking against the store level of X instead of W, i.e.,

$$\mathrm{Trans}_{\mathsf{X}} = \prod D(G\mathsf{T}_{\mathsf{X}}(\Lambda)) \times \prod \left\{ \left\lceil \frac{\mathsf{D_d}}{\mathsf{T_d}} \right\rceil \;\middle|\; \mathsf{d} \in LT_{\mathsf{X}}(\Lambda) \right\},$$

where $\Lambda = FuseSel(\{\mathsf{x}, \mathsf{y}, \mathsf{z}\}, \{\mathsf{x}, \mathsf{y}, \mathsf{z}, \mathsf{i}\})$.

Output Array

Finally, for the number of external memory accesses for array Y, it is easier to deviate from the volume/transfer approach used above. When a layer is fused,

the output simply does not contribute to the external transfers, as the outputs are stored directly in the X array of the layer that is being fused into. When a layer is not fused, eventually the complete output, i.e. $\mathsf{D_x D_y D_z}$ elements, have to be transferred at least once to the external memory. More than one transfer per output element may be required if *partial results* are stored in (and later loaded from) external memory. Here, a partial result is a partial sum in array Y which is not yet ready to be passed to the nonlinear activation function. For internal buffer space it could be interesting to evict some of these partial results from the local buffer, and load them back later. This happens only if tiling is applied to the i loop. In that particular case the partial output elements have to be transferred twice for each tile in i, once for storing the partial results externally, and once for loading them back (excluding the first update of Y). Combined this yields the following expression for the number of elements transferred for output array Y:

$$\mathrm{Acc}_{\mathsf{Y}} = \mathsf{D_x D_y D_z} \times \left(\left\lceil \frac{\mathsf{D_i}}{\mathsf{T_i}} \right\rceil \times 2 - 1 \right).$$

5.4.4 Compute

The final part of the model represents the number of MACs required to complete a schedule. Without recompute, this number is trivial to obtain by multiplying all dimensions of a layer $\mathsf{D_x} \times \mathsf{D_y} \times \mathsf{D_z} \times \mathsf{D_c} \times \mathsf{D_n} \times \mathsf{D_m}$. However, to account for recompute due to overlap of input tiles detailed later in section 5.4.5, this formula is split into three terms: the number of MACs to produce a single output pixel PMACs, the volume of a produced output tile OVol, and the number of such output volumes in a layer.

The number of MACS for a single output pixel is fairly straightforward, and is determined by the number of input feature maps multiplied by the kernel size:

$$\mathrm{PMACs} = \prod K(\{\mathsf{x}, \mathsf{y}\}) \times \mathsf{D_i}.$$

The produced volume in number of features for every transfer is also straightforward, and amounts to those tiles which are below the store level of array X:

$$\mathrm{OVol} = \prod T(GE_{\mathsf{X}}(\{\mathsf{x}, \mathsf{y}, \mathsf{z}\})).$$

Note that the store level of array X is used, since the input volume determines the *produced* output volume. Consequently, the number of such volumes is simply equal to the number of transfers of array X, $\mathrm{Trans}_{\mathsf{X}}$.

The total number of MACS is thus given by:

$$\text{MACS} = \text{PMACS} \times \text{OVol} \times \text{Trans}_{\mathtt{X}}.$$

5.4.5 Layer Fusion

Now that models have been established for the memory footprint, external memory accesses, and number of MAC operations per layer for each of the `W`, `X`, and `Y` arrays in the preceding sections, these models can be combined to provide the same properties for complete sets of fused layers. This is achieved by 'chaining' the provided layer models in a recursive fashion. This is best understood by observing the consumption of `X0` in figure 5.3 used to produce `X1`. Instead of producing complete array `X1`, i.e., $\mathtt{D_{x1}} \times \mathtt{D_{y1}} \times \mathtt{D_{i1}}$ as the output size of `X0`, in a fused schedule only a single tile $(\mathtt{T_x} + \mathtt{D_{k1}} - 1) \times (\mathtt{T_y} + \mathtt{D_{l1}} - 1) \times \mathtt{D_{i1}}$ needs to be produced at a time. Substituting $\mathtt{D_x}, \mathtt{D_y}, \mathtt{D_z}$ with $(\mathtt{T_x}+\mathtt{D_{k1}}-1)\times(\mathtt{T_y}+\mathtt{D_{l1}}-1)\times\mathtt{D_{i1}}$ respectively in the models provides the costs of this intermediate tile:

- *Multiply accumulates:* The number of MACs required to produce the tile from `x0` to `X1` are simply given by PMACs($L01$), where $L01$ represents the layer that consumes `X0` and produces `X1`.
- *Memory footprint:* The local memory footprint of the tile in `X0` is given by $\text{Buf}_{\mathtt{X}}(L01)$, the required footprint of the tile in `X1` is given by $\text{Buf}_{\mathtt{X}}(L12)$. The footprints of the weights can used without any substitution.
- *Accesses:* The intermediate tile in `X0` of course does not require any external data accesses, as it is produced and consumed in a fused fashion. However, the number of transfers $\text{Trans}_{\mathtt{X}}(L12)$ specify how many times the tile needs to be produced in the case of recomputation, and tile overlap in general. For the overall cost of the fused set of layers, the MACs for the tile are to be multiplied with the number of productions. Same holds for the loads of the weights required to produce the tile, unless they are completely stored in the local buffer. Also, this number of required productions moves further up the set of fused layers, as consequently any producers of this tile may in turn need to be produced multiple times if they are not stored for the complete lifetime of the network depending on their store level and tile sizes. Only the first layer of a set of fused layers, has external memory accesses. If this first layer needs multiple transfers of its own, i.e., not all uses of the data elements are captured on first load, the number of additional accesses can grow quite quickly due to the recomputation effect. The more profitable schedules therefore typically ensure the input is

loaded only once, such that the cost of recomputation indeed only affects the compute cost, and does not increase the number of external accesses.

Hence, using substitution of tile dimensions for the output dimensions, and propagation of the number of productions required for each tile, the total cost of a fused segment can be derived from the individual cost models presented in the previous sections.

5.4.6 Complete Network Model

The total costs of a complete network are now trivial. The network schedule effectively partitions the layers into groups of fused layers. These groups contain one or multiple layers, for which the costs can be derived using the fusion approach described in the previous section. The total number of multiple accumulates required by a network schedule is simply obtained by adding the MACs of all groups. Same holds for the number of external accesses, the sum of the accesses of each group forms the cost for the entire network. The memory however only requires the maximum buffer size over all groups, since only a single group is active at any given time during network evaluation assuming no pipelining of the execution. Using these simple rules, the costs of an entire network can be computed.

5.5 Automated Design Space Exploration

The formal model introduced in section 5.4 enable automated exploration of the vast scheduling space described in section 5.3. To achieve complete automation section 5.5.1 describes a strategy to efficiently traverse the entire scheduling space. Section 5.5.2 introduces `ConvFuser` [154], an open source tool which implements the presented traversal strategy and cost models, enabling automated design space exploration and verification of any neural network described in the popular Keras framework [23].

5.5.1 Space Traversal

To effectively explore the scheduling space as described in section 5.4 four steps are required:

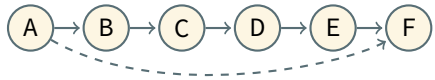


Figure 5.4: Example layer graph with sequence `[B,C,D,E]`, and a residual connection between `A` and `F`.

Identification of sequences

Since the proposed models do not include provisioning to handle residual/skip connections, only consecutive layers without forks or joins are considered for fusion. The first step of a design space exploration (DSE) is therefore to select sets of layers that *may* be fused. Such a set of eligible layers is referred to as a *sequence*, an example of which is shown in figure 5.4.

Segmentation

Within each sequence it needs to be decided which layers to fuse (if any) to obtain the Pareto optimal schedules of the network. A set of fused layers within a sequence is referred to as a *segment*. The *sequence* `[B,C,D,E]` of figure 5.4 contains the following valid *segments*: `B, C, D, E, BC, CD, DE, BCD, CDE, BCDE`. Each of these segments is evaluated individually, yielding a vector of schedules $\boldsymbol{S}$ per segment.

Partitioning

Once all possible segments have been identified and their costs have been evaluated, those segments that cover the entire sequence need to be combined. For example segments `BC` and `DE` cover sequence `[B,C,D,E]`, but also segments `B, C,` and `DE`, as well as many more. Combining the schedule vectors $\boldsymbol{S}$ of each segment into an overall schedule vector $\boldsymbol{SS}$ for the sequence is done by taking their product, and using the rules outlined in section 5.4.6. Note that this procedure can be significantly accelerated by first Pareto-filtering the segment schedule vectors $\boldsymbol{S}$. It can be trivially proven that considering only the Pareto points of each segment is sufficient to yield all the Pareto points of the entire partition, since the combining (reduction) functions of section 5.4.6, i.e., summation and maximum selection, are monotonically increasing.

Network Schedule Cost Computation

Finally the costs of all valid partitions are combined using the same rules of section 5.4.6 to obtain the cost of the entire network. Similarly to the partition cost computation, segment schedule vectors $\boldsymbol{SS}$ can be Pareto-filtered before combination with other partitions to yield a Pareto optimal scheduling of the entire network.

5.5.2 ConvFuser

An embodiment of the automated design space exploration described in this chapter is provided in the form of an open source tool: `ConvFuser` [154]. Apart from automated DSE, `ConvFuser` also features code generation for any selected schedule, enabling reliable validation of the models.

`ConvFuser` builds upon the popular Keras/TensorFlow framework [23] to read standard HDF5 graph models (See figure 5.5). After loading a network using Keras/TensorFlow, a graph is constructed from the network layers, and a custom canonicalisation pass is employed to normalize the network description. An important part of this canonicalisation is performing trivial layer merges, including but not limited to:

- merging of batch normalization layers into convolutional layers, which can be achieved by modification of the weights of the target convolutional layer.
- merging of activation layers into convolutional layers.

Note that some literature refers to these trivial layer merges as layer fusion. This term is apt in the case of merging activation layers, which also involves loop fusion, but it should not be confused with the much more complicated fusing of consecutive convolutional layers as described in this chapter.

After canonicalisation, design space exploration can be performed. Many smart search strategies could be employed here, but by virtue of the mathematical models and their fast evaluation, straightforward exhaustive search is feasible for smaller networks. Additionally the tool provides several options to restrict the design space, such as limiting the number of layers considered for fusion, selection of tile sizes such as only exact multiples of their respective dimension, or only powers of two, and whether to consider recomputation.

Finally, to enable validation of the found schedules, a hybrid back end based on the Halide language [124] and Keras/TensorFlow [23] is provided. Any non-

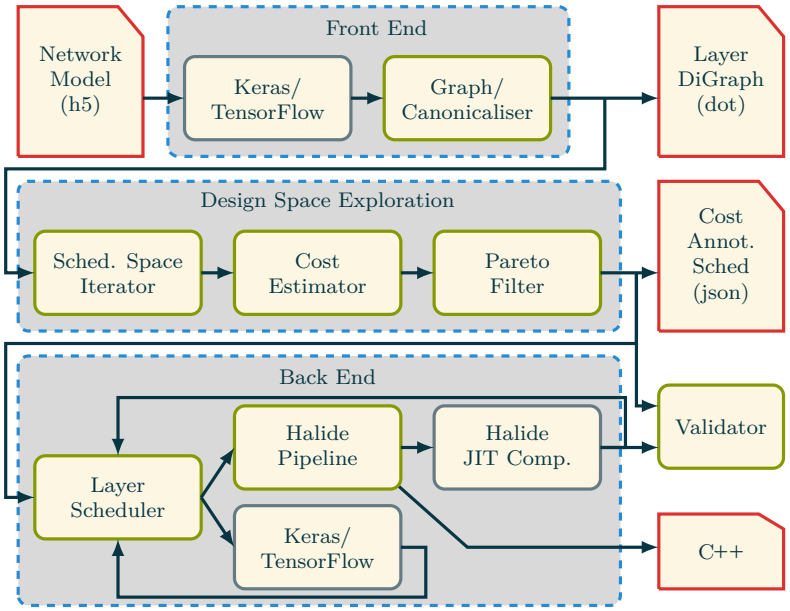


Figure 5.5: Schematic overview of the `ConvFuser` tool. Green boxes are developed for this chapter. The tool consists of a Front-End which translates keras/tensorflow networks into an internal representation, a design space exploration component which uses the models introduced in this chapter, and a Back-End which generates Halide and C++ code for the selected schedule. Source code is made freely available [154].

convolutional layers are evaluated directly by Keras/TensorFlow. The scheduled convolutional layers however are implemented using a modified version of Halide. In particular this modification consists of additional python bindings to be able to insert instrumentation code. This code utilizes Halide's internal tracing mechanism to keep track of accesses to buffers, and sizes of allocated buffers. To emulate an external memory and internal buffers, a construct similar to Halide's recently added `in` operator is used. This construct adds an extra layer of buffering, where one large buffer essentially mimics external memory, and working data is loaded into a smaller, internal buffer. By keeping track of the accesses to these two levels of buffering, the required external accesses can be exactly monitored. An optional validation step can be used to check the external accesses, internal buffer size, and MACs measured from Halide execution with the values predicted by the models.

5.6 Model Validation & Evaluation

The validity of the models introduced in section 5.4 is confirmed experimentally using the `ConvFuser` tool introduced in section 5.5. The experiments consist of a design space exploration for several synthetic and real-world networks, followed by *code generation* for each of the Pareto optimal schedules. This code is automatically instrumented by the `ConvFuser` tool to measure the number of MACs, required internal memory size, and number of external accesses, which are then compared against the modelled values.

Besides this model validation, the tool also enables the evaluation of the impact of various scheduling techniques. In essence, the scheduling space described in section 5.3 can be restricted to subsets by disallowing or limiting certain scheduling techniques. For this evaluation four progressively inclusive scheduling spaces are defined:

1. *Baseline*: The baseline scheduling space follows the straightforward implementation in code 5.1 for each layer. It thus excludes loop reordering, tiling, and fusion, but does allowing different store and compute levels.

2. *Tiling & Reorder*: This space adds both loop tiling and reordering to the allowed options to the baseline space. Tiling is restricted to sizes that are integer factors of the dimensions to keep the space to a size that can be completely traversed in reasonable time. Other tiling options are built into `ConvFuser`, but are not further evaluated in this section.

3. *Goetschalckx et al.*: This space extends the tiling and reorder options with layer fusion. The allowed fusion however does not recompute any elements. Furthermore weights are not tiled, and are always stored on-chip for a fused section. This space matches the work of Goetschalckx et al. [40], enabling direct comparison. One notable exception for this space is that the tiling factor is not limited as is the case in the work of Goetschalckx et al., since for this particular design space it is possible to use a fast branch and bound algorithm on the tiling factor while still guaranteeing an optimal result.

4. *Fusion & Recomputation*: This extends the *Goetschalckx et al.* space with tiling of weights and recomputation. The addition of tiling the weight accesses however disallows the previously mentioned branch and bound technique on the tiling factor without losing optimality. Therefore this space again is constrained to the same tiling limitations as the *Tiling & Reorder* space, i.e., integer factors of the dimensions.

This partitioning of the scheduling space enables the investigation of the impact of tiling, loop reordering, fusion, and recomputation on MAC count, required on-chip memory, and external accesses.

5.6.1 Micro benchmarks

In order to validate the models introduced in section 5.4, and obtain insight into the effect of different scheduling techniques, first a number of micro benchmarks is performed. To this end two synthetic networks, L2Net and L3Net, are defined. These networks consists of respectively two or three convolutional layers with a 3×3 kernel. This enables a study into the effects on the resulting scheduling space when deepening networks. The dimensions of these layers are shown in table 5.6 for both a 20×20 input and a $4K$ input, used to see the effects of network input size on the resulting schedules. Finally the number of feature maps of these two nets is increased, yielding L2NetWide and L3NetWide in table 5.6, to examine the effect of this parameter on the effect of the modelled scheduling techniques.

The results of the schedule space exploration for L2Net and L3Net with 20×20 and 22×22 inputs respectively are shown in figure 5.6a figure 5.6b. For all schedules of a network the number of MACs is constant, with the exception of those schedules that include recomputation. To indicate the number of required MACs, a colour coding is added for the *Fusion & Recomputation* schedules. The

Table 5.6: Layer dimensions of L2Net and L3Net for 20×20 and 4K inputs.

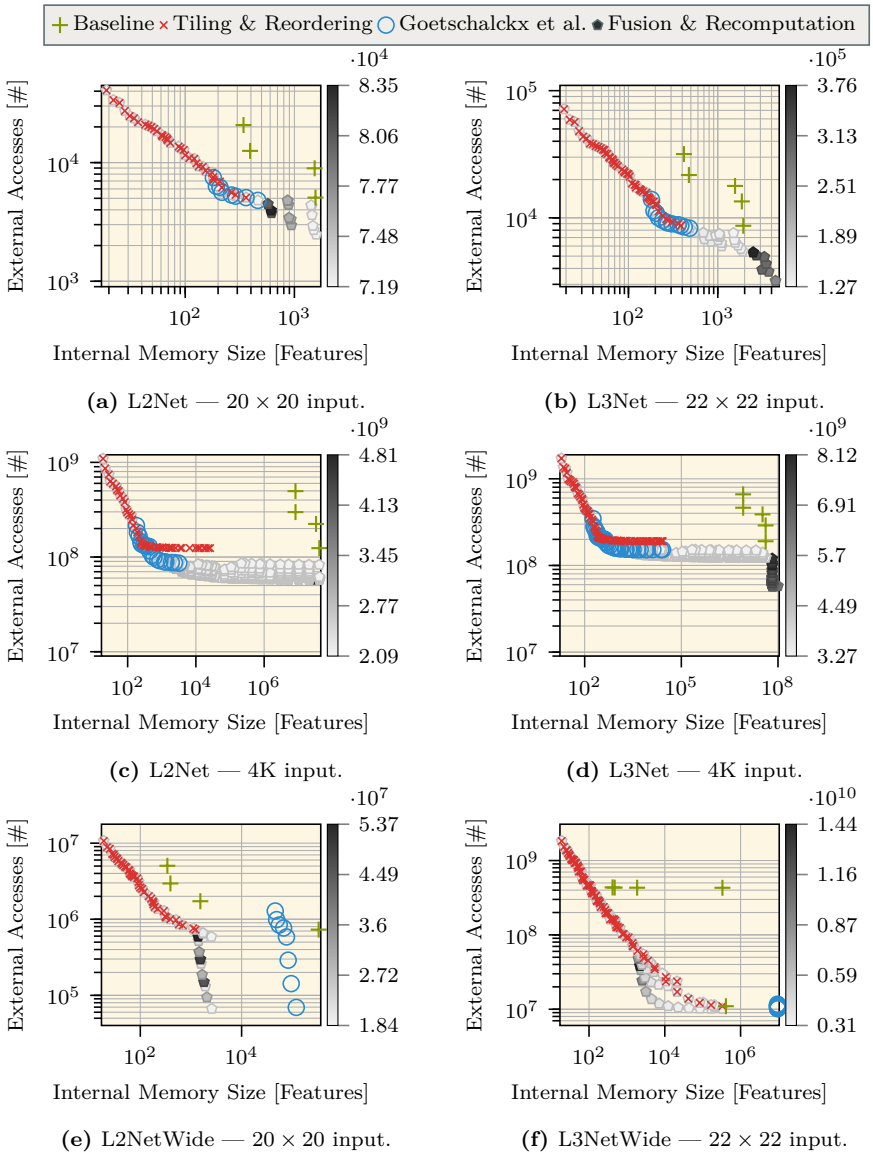
Net	D_x	D_y	D_i	D_z	D_n	D_m
L2Net	18	18	3	4	3	3
	16	16	4	4	3	3
L3Net	20	20	3	4	3	3
	18	18	4	4	3	3
	16	16	4	4	3	3
L2Net 4K	3838	2158	3	4	3	3
	3836	2156	4	4	3	3
L3Net 4K	3838	2158	3	4	3	3
	3836	2156	4	4	3	3
	3834	2154	4	4	3	3
L2NetWide	18	18	3	1024	3	3
	16	16	1024	4	3	3
L3NetWide	20	20	3	1024	3	3
	18	18	1024	1024	3	3
	16	16	1024	4	3	3

Pareto schedules for L2Net and L3Net with 4k inputs are shown in figure 5.6c figure 5.6d respectively. Finally the Pareto fronts for L2NetWide and L3NetWide are given in figure 5.6e figure 5.6f. Note that the verification with instrumented Halide for the selected points in figure 5.6 showed a *one to one match* between the models and the measurements, apart from some corner cases where Halide failed to apply complete buffer folding. These results confirm the accuracy of the models presented in section 5.4. The measured and modelled points have not been plotted together, since their points would simply overlay each other.

From these figures five key observations can be made:

1. The baseline schedules are consistently poor over all networks, although they always include a point that minimizes the external accesses when fusion and recomputation are not considered. This naturally happens for the store-level that cover all data uses, but due to the lack of tiling and smart loop reordering these points typically require a huge amount of memory.

2. Adding tiling and loop reordering on top of the baseline schedule results exclusively in schedules that require less internal memory, and thus completely Pareto dominate the baseline points.



(a) L2Net — 20 × 20 input.

(b) L3Net — 22 × 22 input.

(c) L2Net — 4K input.

(d) L3Net — 4K input.

(e) L2NetWide — 20 × 20 input.

(f) L3NetWide — 22 × 22 input.

Figure 5.6: Pareto schedules of the synthetic networks for the four defined scheduling spaces. The colour map only applies to *Fusion & Recomputation*, and represents the number of MAC operations.

3. Fusion without recomputation and weight tiling, i.e., the space defined by Goetschalckx et al. [40] does not add many interesting points for the small input versions of L2Net and L3Net. The accesses saved by omitting transfers on the small intermediate layers does not add much for these small networks where the number of weights are relatively significant. When applied to the 4k input networks however, some gains can be observed as the weight transfers become insignificant compared to the data transfers. However, the gains are still rather modest since the remaining transfers on the input layer have also scaled with the input resolution. More interesting are the points for L2NetWide, where fusion does yield larger gains as the number of feature maps increases, and the intermediate transfers start to dominate the remaining input transfers. However, as the weights are again important in these networks, the ideal combination for the Goetschalckx scheduling space, i.e., large inputs and wide intermediate layers, is not achieved in these synthetic benchmarks.

4. Enabling weight tiling and recomputation however does yield some interesting points. In particular for the wide networks with small input, i.e., those networks where weights are significant, some schedules with even modest recomputation are found to outperform the Goetschalckx schedules by an order of magnitude in required memory size, demonstrating the added value of the more generic models presented in this chapter.

5. As the input size and number of layers increases, recomputation becomes more and more relevant, but only for very large memory sizes. Apart from these extreme cases, the memory saved by heavy recomputation is rather marginal compared to neighbouring schedules with minimal to no recomputation.

5.6.2 Real World Networks

Although above synthetic micro benchmarks provide insight into general trends, it is important to also evaluate the scheduling techniques in the context of real world networks. To this end six widely used networks are selected for experimentation: ResNet50, VGG16, InceptionV3, MobileNetv2, and XCeption as implemented in the Keras framework [23], and DMCNN-VD, the demosaicing as described by Syu et al. [144] and also evaluated by Goetschalkx et al. allowing for direct comparison. The scheduling Pareto fronts of these networks are given in figures 5.7a to 5.7d and 5.7f respectively. Because the complete design spaces of these networks are exceptionally large, an *exhaustive* traversal of the design

space is infeasible even with the presented fast models. Therefore the exploration of the design space of these networks was limited to ten million considered schedules per *segment*[2]. This limitation particularly impacts the *Fusion & Recomputation* space, since it is the largest of the four spaces. As a result the Pareto fronts are less smooth than those of the micro-benchmarks which are exhaustively searched. Nonetheless, the general trends can be easily observed, and plenty of schedules remain on the Pareto fronts for practical purposes. From figure 5.7 the following three observations are made:

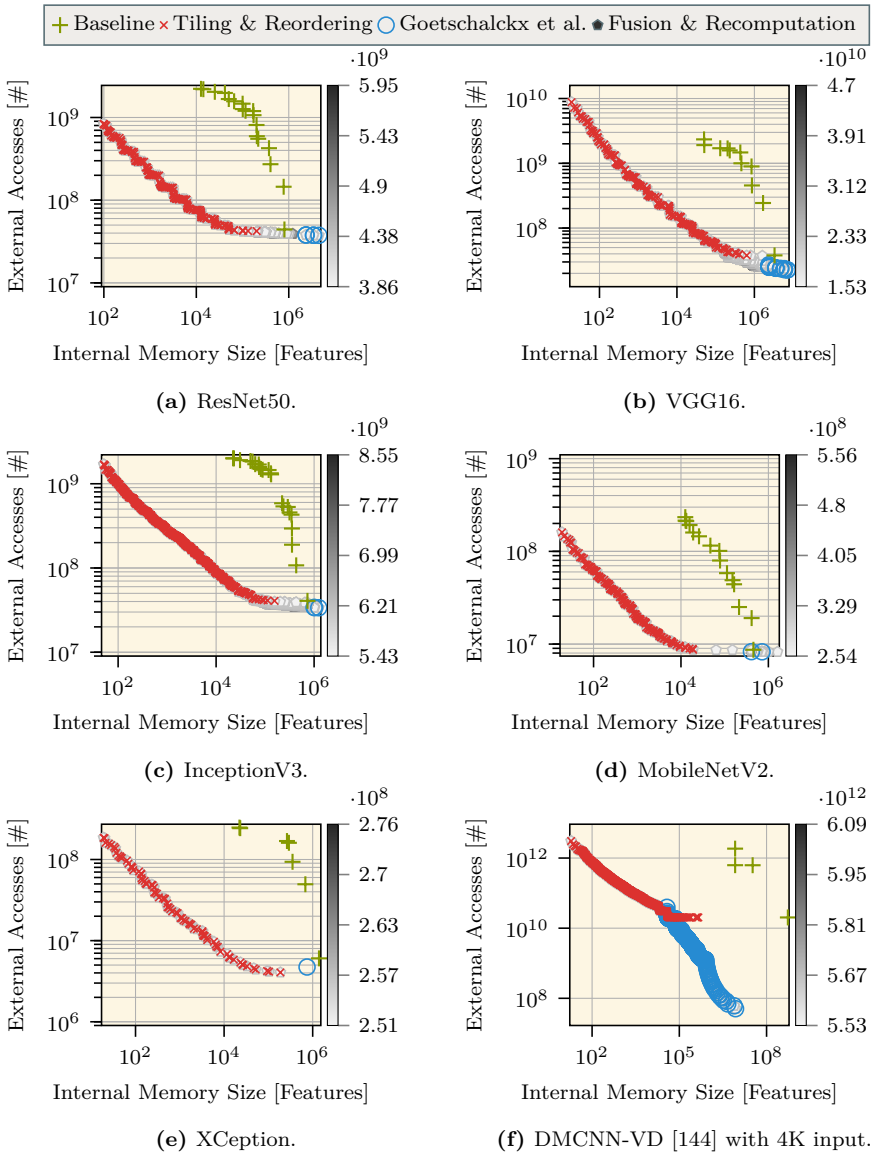
1. For typical real-world networks with 224×224 ImageNet input resolutions, the gain of fusion and recomputation on top of tiling and reordering is rather limited. This effect is expected based on the micro-benchmarks, in particular keeping in mind that in real-world networks various layer types prohibit the application of fusion with the presented models, limiting the benefits of fusion even further. Only for very large memories do the accesses decrease.

2. For networks with large input, and a straightforward structure of sequential convolution layers such as DMCNN-VD, the story is slightly different. Here extra memory can effectively be used for fusion to reduce the number of external accesses significantly.

3. The *Fusion* Pareto front of DMCNN-VD matches the experiments of Goetschalckx et al. [40], further validating the more generic models presented in this chapter. Moreover, more Pareto schedules are found as the tiling factor was not limited in our experiments by virtue of a branch and bound design space strategy on the tiling factor which still guarantees optimality. This enabled exploration of the complete scheduling space of DMCNN-VD as defined by Goetschalckx et al. [40] in a matter of minutes.

Based on comparison with the generated, instrumented code, it can be concluded that the presented models are very accurate. Their added value over the state of the art has been demonstrated, in particular for networks where the number of weights is significant. Although for real-world networks the benefits of fusion are somewhat limited due to complex connections and various layer types, the external accesses can in most cases be reduced compared to loop reordering and tiling only. The best observed reduction in external accesses was as high as 99.75 % for DMCNN-VD. On average the gain of fusion over tiling and loop reordering was 28.89 % for the six selected networks.

[2]See 'segmentation' in section 5.5 for the precise definition of a segment.



(a) ResNet50. **(b)** VGG16.

(c) InceptionV3. **(d)** MobileNetV2.

(e) XCeption. **(f)** DMCNN-VD [144] with 4K input.

Figure 5.7: Pareto schedules of six Real-World networks for the four defined scheduling space. The colour map only applies to *Fusion & Recomputation*, and represents the number of MAC operations.

5.7 Energy Consumption

The scheduling results presented in the preceding section expose the potential to reduce the external memory accesses using advanced scheduling techniques. This section extends these results with a short study into the effects of this reduction in accesses, since for most practical designs not the raw accesses, but the energy consumption is of interest. Building upon the metrics produced by the models introduced in this chapter, the remainder of this section introduces progressively more realistic energy models starting from a single level, single bank SRAM internal memory in section 5.7.1 till a multi-level, multi-bank internal memory model in section 5.7.3

5.7.1 Single-Bank SRAM

The energy required to evaluate a neural network according to a specific schedule can be split into three distinct parts:

Multiply-Accumulate Operations

The first source of energy consumption is the execution of MAC operations. The energy required for a single MAC depends on the data type of the operation, the arithmetic architecture, the operating speed, and the technology node used for implementation. How these parameters influence the energy consumption is complicated, and instead of attempting to derive a generic model choices are made for these parameters in this evaluation. In particular, a 40 nm node is assumed for which energy numbers are available in the Aladdin tables [176], which are also used by the state-of-the-art Accelergy energy estimation tool from MIT [174]. In alignment with the accelergy framework, the cost of a multiplication and addition at 4 ns delay are summed to conservatively approximate a MAC unit. Although the model metrics are agnostic to the width of a neuron output or weight value, a choice has to be made to be able to provide an energy estimate. Since the listed numbers are for 32b operations, and the operations in neural networks usually only require 16b, a scaling factor is applied. Specifically, the energy of the addition is halved since adders scale approximately linearly, and the multiplication energy is divided by four because of quadratic scaling of multipliers. Following this method, the energy cost of a single MAC operation is approximated at 10.2 pJ.

Accessing External DRAM

The second factor influencing the energy consumption is accessing external memory. DRAM is a typical choice for off-chip memory, and is also assumed in this evaluation. According to the work of Malladi et al. [100] accessing a single bit in DDR3 memory requires about 70 pJ. For simplicity it is assumed that both the neuron outputs and weights are 16-bit wide, yielding an energy cost of 1120 pJ per DDR3 DRAM access. Note that this number is taken to be independent of the DRAM size, since the energy is dominated by input/output (IO) logic rather than the size of the memory array.

Accessing Internal SRAM

For internal memory SRAM is assumed. Estimating the energy of an SRAM access is slightly more involved, since the size of the memory array does matter significantly for the access energy. To be able to provide accurate estimates, a model is derived based on commercial-off-the-shelf SRAM modules on a 40 nm technology node. Figure 5.8 shows available data points for these modules in terms of access energy for 16b words based on the total module size in bits. These numbers are based on the typical corner, 25 °C, 1.1 V, averaging a read and write access. A square root function is fit through these points to enable extrapolation, yielding the energy per access of an s-bit SRAM cell in pJ:

$$\mathrm{E}_{\mathrm{SRAM}}(s) = 0.012 \cdot \sqrt{s} + 4.61$$

A square root function is chosen as the energy cost of larger banks mainly seems to scale with the circumference of the array, and the additional sense amplifiers used to partition the bank internally. This model is used to extrapolate the energy cost of an SRAM bank in figure 5.8. Note that the range of the extrapolation is rather extreme to be able to support the large memories required for some of the fusion schedules. This already indicates a different approach to constructing such large memories should be taken in practice, which will be further elaborated in section 5.7.2. Although the selected square root function fits the relation quite well (see also figure 5.10 which clearer shows the fit through the available data-points), this model can easily be exchanged for more sophisticated and accurate models when available.

Adding these three factors, i.e., MAC energy, DRAM energy, and SRAM energy, yields the total energy estimated by this basic model. For the number of MACs the `MAC` numbers from the model can be directly used, the accesses to DRAM are also directly given by the various `Acc` terms, as is the size of the internal SRAM

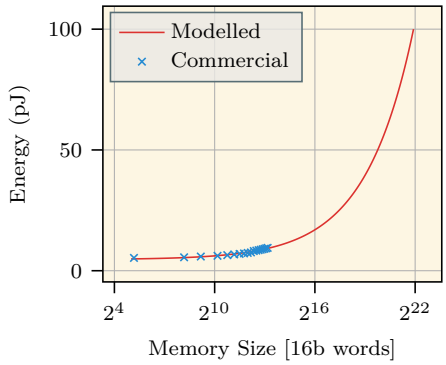
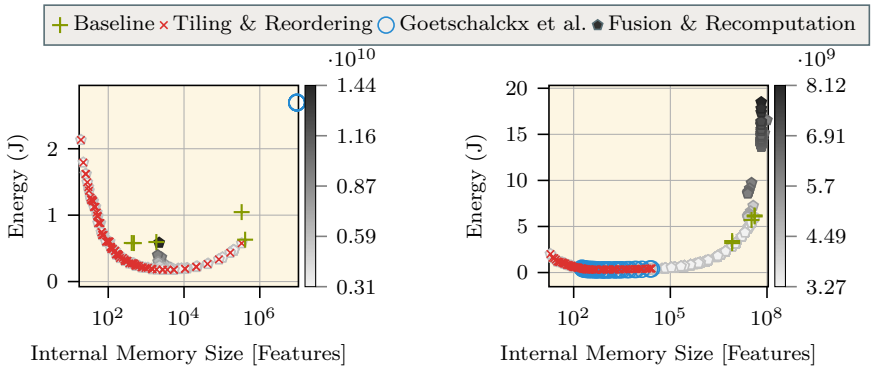


Figure 5.8: SRAM energy consumption based on commercial 40 nm SRAM modules at the typical corner, 25 °C, 1.1 V. Power quickly grows for large SRAMs blocks.

by taking the maximum of the required `Buf` formulas over all layers. This leaves the number of accesses to the SRAM however, which are given by the number of MACs times four, since each MAC operation requires three reads and one write. Note that this basic model lacks a register file or accumulation register, and hence all accesses go to the internal SRAM. Finally each external read access will also write to the internal SRAM, so these accesses are also added.

Applying this model to the scheduling front of the L3NetWide and L3Net 4k networks presented in section 5.6 yields figure 5.9. Since the energy model is monotonically increasing in all parameters, the Pareto schedules found in section 5.6 are guaranteed to contain those schedules that are also Pareto in terms of energy. The energy points are not Pareto filtered in this case however to highlight an important drawback of the used energy model. Using the simplistic $\mathrm{E}_{\mathrm{SRAM}}(s)$ approximation, the internal SRAM memory quickly becomes very expensive to access as its size increases. In particular, past around 5×10^3 features the reduction in external DRAM accesses no longer outweighs the increased cost of accessing the internal SRAM, and the energy starts to increase again. The problem is that the access energy of a single SRAM bank does not scale very well, and for those schedules that require a large amount of internal memory a more sophisticated memory model is required.



(a) L3NetWide. **(b)** L3Net 4k.

Figure 5.9: Energy front of L3Net based models using the single bank SRAM model outlined in section 5.7.1. The colour map only applies to *Fusion & Recomputation*, and represents the number of MAC operations.

5.7.2 Multi-Bank SRAM

The shortcoming of the basic energy model is the assumption of a single SRAM bank also for large internal memories. In practice, however, such large memories will always be constructed out of several smaller SRAM banks. Such memories will incur an area penalty compared to the single bank approach, but the access energy is lowered significantly. In fact, the access energy is equal to the access energy of a single small bank that makes up the larger memory, plus some overhead for the bank selection logic. To correct for this energy overhead a scaling function is fit based on the work of Mai et al. [99]. To minimize the impact of the area overhead, and provide a pessimistic estimate of what can be achieved with a banked SRAM memory, the largest available SRAM block from the commercial 40 nm library is selected. Combined with the energy overhead function this yields the following model for the access energy of an s-bit multi-bank SRAM memory:

$$\mathrm{E}_{\mathrm{Banked_{SRAM}}}(s) = \mathrm{E}_{\mathrm{SRAM}}(\min(s, 16 \times 10^4)) \times \left(1 + 3.05 \times 10^{-3} \times \log\left\lceil \frac{s}{16 \times 10^4} \right\rceil\right)$$

This relation is visualized in figure 5.10, from which it is clear that the overhead of the bank selection logic indeed is relatively small. Plugging this multi-bank

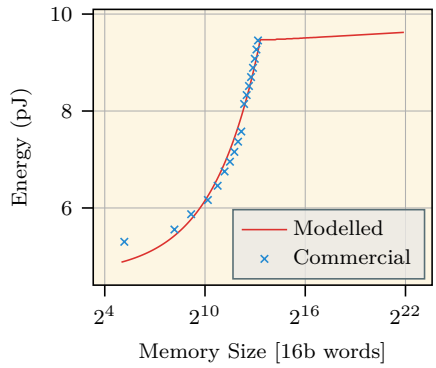
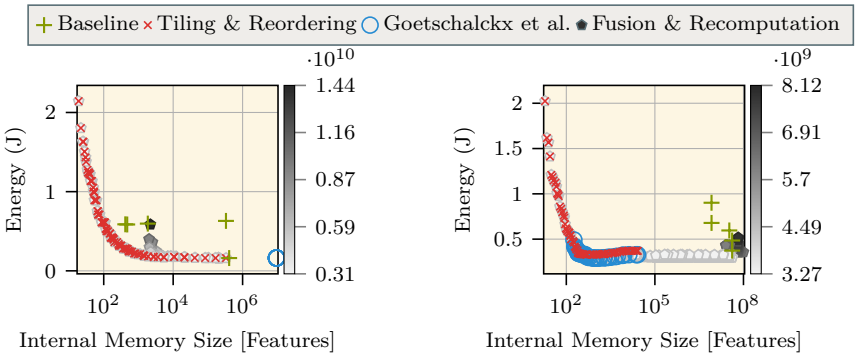


Figure 5.10: *Banked* SRAM energy consumption based on commercial 40 nm SRAM modules at the typical corner, 25 °C, 1.1 V. Large memories are constructed by replicating the largest available SRAM block to create several banks, rather than extrapolating the energy of a single block.

SRAM model into the energy model described in section 5.7.1, and again applying it to the L3NetWide and L3Net 4k networks yields figure 5.11. It is immediately clear for this figure that the multi-bank SRAM model largely solves the energy problem for schedules that require large amounts of internal memory. For L3NetWide the problem is gone entirely, while for L3Net 4k the optimum memory size still lies around 1×10^3 entries. In both cases it is clear however the network schedules with larger internal memory requirements hardly benefit from the reduction in external accesses, even while accessing the DRAM is relatively expensive. Furthermore all schedules with recomputation always increase the energy consumption. Both these observations can be explained by the lack of a register file or other small localized memory, and as such each additional MAC incurs four accesses to the internal SRAM.

5.7.3 Multi-Level SRAM

To overcome the limitation of a single level internal SRAM memory, this section expands the energy model with multi-level internal memory. This does present a problem, however, since the models presented in this chapter do not support multi-level memories in a precise manner. Section 5.8 contains notes on how such support may be added as part of future work, but for this evaluation an approximation is used instead. In particular, when a memory level of size s is



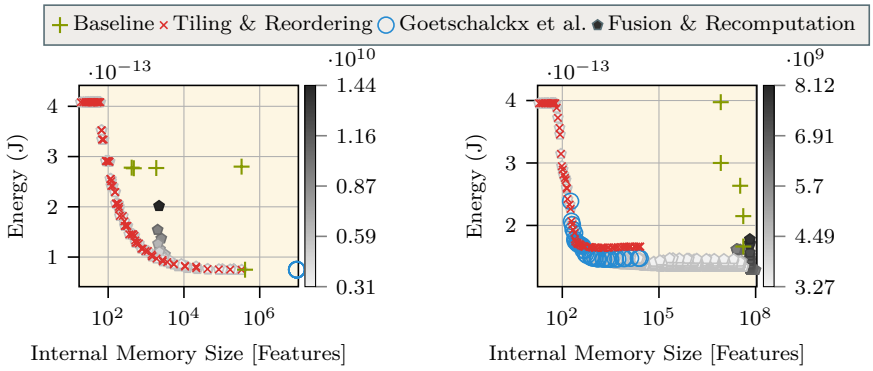
(a) L3NetWide. **(b)** L3Net 4k.

Figure 5.11: Energy front of L3Net based models using the multi-bank SRAM model outlined in section 5.7.2. The colour map only applies to *Fusion & Recomputation*, and represents the number of MAC operations.

added, the best schedule available for that size is used to model the accesses to that level. Furthermore the 'external accesses' modelled for this schedule will now be added to the next level of memory. This next level is determined by a sweep over the remaining schedules, and using their internal memory size for this level. Using this methodology a register file with $64 \times 16b$ entries is added to the energy model. For accessing this register file an average accesses energy of 2.4 pJ is used in accordance with the findings of Wu et al. [175].

The resulting energy fronts of the L3NetWide and L3Net 4K networks are given in figure 5.12. It can be seen that the addition of the register file ensures that the energy does decrease when larger internal memory is used. Also in L3Net 4k some of the schedules with recomputation have become beneficial, although the effect is rather marginal.

In the real-world networks evaluated in section 5.6 similar trends can be observed. For brevity only the energy fronts of VGG16 and DMCNN-VD are shown in figure 5.13. Similarly to L3NetWide and L3Net 4k, layer fusion does result in new energy Pareto schedules, although the benefits are not as large as the reduction in external memory accesses in figure 5.7 imply. This is in particular clear for the DMCNN-VD network, which has a significant reduction in DRAM accesses using layer-fusion (figure 5.7f), but fails to capitalize on this in the energy front (figure 5.13b). Furthermore, recomputation in the real-world networks



(a) L3NetWide. **(b)** L3Net 4k.

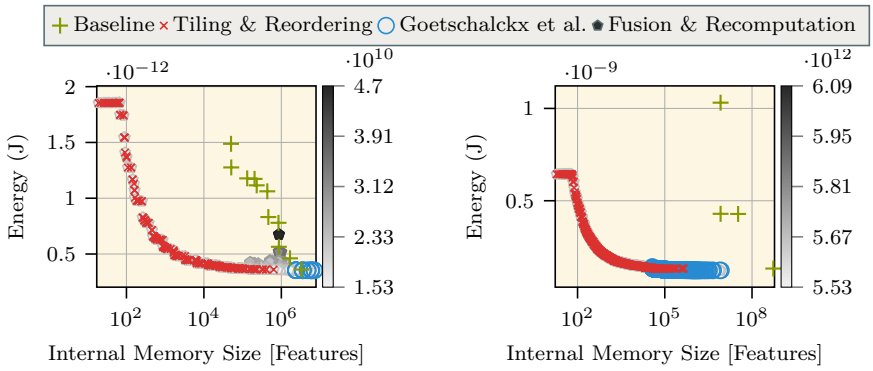
Figure 5.12: Energy front of L3Net based models using the multi-bank SRAM model outlined in section 5.7.2 and a $64 \times 16b$ register file. The colour map only applies to *Fusion & Recomputation*, and represents the number of MAC operations.

does generally not result in a reduction in energy, as can be seen for VGG16 in figure 5.13a.

The limited gains of fusion may seem counter-intuitive at first, in particular for the DMCNN-VD network which shows significant reduction in external memory accesses in figure 5.7f. These reduced returns can easily be understood when inspecting the detailed energy breakdown in figure 5.14 however, which lists the energy spent on MACs (E_{mac}), the register file (E_{rf}), the internal SRAM (E_{sram}), and the external DRAM (E_{dram}), for each point in the energy Pareto front. The reason the reduction in accesses does not result in a significant drop in energy consumption is effectively Amdahl's law applied to energy saving; As the scheduling techniques reduce the amount of accesses to the DRAM, the other parts of the system start to dominate. For the overall energy to improve even further, it makes most sense to address the E_{mac} and E_{rf} components by for example chaining units to avoid intermediate accesses to the register file, dedicated accumulation registers, and quantization of the MAC operands.

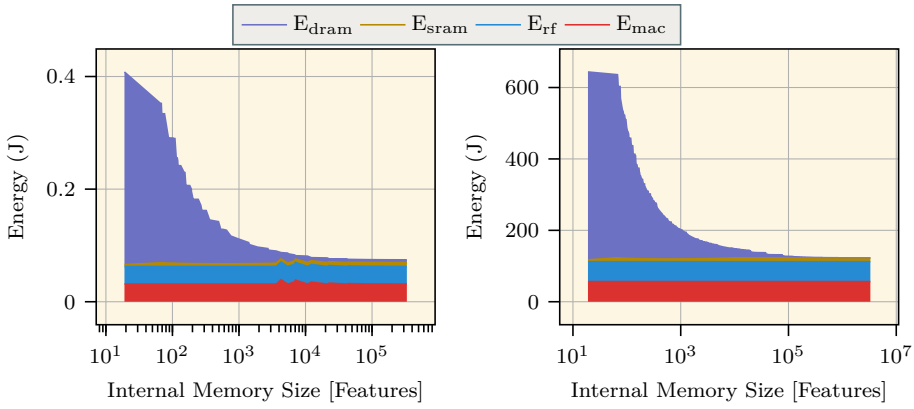
In conclusion the following statements can be made regarding the energy consumption of the explored schedules:

1. Compared to the baseline schedules, many schedules that require less internal memory can be found which drastically reduce the energy consumption.



(a) VGG16. **(b)** DMCNN-VD.

Figure 5.13: Energy fronts of VGG16 and DMCNN-VD. The colour map only applies to *Fusion & Recomputation*, and represents the number of MAC operations.



(a) L3NetWide. **(b)** DMCNN-VC.

Figure 5.14: Detailed energy breakdown of energy Pareto front. For large SRAMs, the DRAM energy reduces drastically, while the register file (RF), MAC, and SRAM energy remain roughly constant.

2. Based on the energy evaluation a multi-banked and multi-level memory approach is required to benefit from the gains of fusion and recomputation, and even then energy gains are not always guaranteed.
3. The reduction in external accesses achieved by advanced scheduling techniques such as recomputation and fusion do not automatically translate in large improvements in energy, as other parts of the system become dominant in this region of the scheduling space for the investigated neural networks.

5.8 Discussion & Open Issues

By validation with instrumented Halide code, the introduced models are shown to be correct for the micro-benchmarks when the layer dimensions are exact multiples of the selected tile sizes. However, despite the low computational complexity of the models, complete traversal of the scheduling space is still infeasible for larger networks. This issue is discussed in further detail in section 5.8.1.

The models presented in this chapter are to a high degree hardware agnostic. A downside of this approach is that certain schedules that look beneficial using these models may be a bad fit on a particular target machine. Section 5.8.2 discusses approaches on how to adjust the design space to match with real machines.

Finally section 5.8.3 discusses several scheduling space limitations of the current work, and possible ways to improve.

5.8.1 Intelligent Design Space Exploration

For small networks the presented models are sufficiently fast to enable an exhaustive schedule space exploration. However, in particular for networks with larger eligible *sequences*[3], the design space grows exponentially. This chapter does not provide a ready solution to this problem, but some suggestions can be made.

The presented models run in constant time per layer, so there is not much to be gained by simplifying the models. Multithreading could be added since the cost estimations are largely independent, but this will only provide a linear improvement against the exponential growth in workload. The provided open

[3]See 'Identification of sequences' in section 5.5 for the precise definition of a sequence.

source tool [154] supports limiting the fusion depth, which can be used to mitigate the problem. As mentioned in section 5.6, the exploration of the real world networks was limited to ten million schedules per *segment*. Still the search completed within a maximum of two days for the selected networks using a fairly unoptimized python implementation of the models, running with only a single thread. Since proper training of a neural network typically takes much longer, and selecting the schedule only needs to be done once before deployment, this may an acceptable runtime for many practical cases. Based on the found scheduling spaces, it also seems unlikely a complete exploration would yield significantly better schedules.

Nonetheless, for particularly large networks, or when schedule quality is extremely important, this approach may not be desired. For those cases two suggestions can be made:

Branch & Bound

When an upper and or lower limit for the internal memory size is known, a 'branch and bound' search strategy can be used to quickly eliminate large sections of the design space. For example, if a schedule with a certain tile size does not fit the available memory, the same schedule with a larger tile sizes will by extension also not fit. Thus, by placing restrictions on the memory size, it is possible to quickly eliminate large parts of the design space.

Heuristics

If the search space can not be sufficiently limited, it is recommended to integrate the presented models into a heuristic based search strategy. For example simulated annealing, genetic algorithms, or a greedy approach with handcrafted heuristics. Another interesting approach is to use an artificial intelligence driven search. Since the models are fast to evaluate, a large training set can be made relatively quickly.

5.8.2 Targeting Real Hardware

The models presented in this chapter are almost completely hardware agnostic, as the only assumption made is the presence of a two level memory hierarchy. However, the `ConvFuser` [154] tool automates (C++) code generation for any schedule within the design space, which allows easy compilation towards any target with a C-compiler. However, the generated code will have no specialisation

towards the target hardware, and therefore either relies on a smart compiler, or (manual) source-to-source transformations. The generality of the models may even seem to adversely impact their applicability to real hardware which supports features such as burst-transfers, and vectorisation. However, targeting such hardware can be easily achieved by restricting the complete scheduling space as described in section 5.3. The remainder of this section discusses how to restrict the space with respect to burst transfers, vectorisation, and how to extend the models to multiple memory levels. Furthermore it treats the possible extension of the models to include throughput and latency, two hardware dependent properties.

Burst Transfers

Burst transfers can amortize the addressing and synchronisation overhead of memory accesses when the data is accessed in larger consecutive blocks. Without adaptation the scheduling space will include schedules that access only one memory element per transfer, and accesses consecutive in time may not at all be consecutive in memory. These schedules may look promising based on number of transfers and required memory size, but due to their inability to benefit from burst transfers may in fact be worse than schedules that did not seem beneficial. The straightforward method of dealing with this is to exclude such non-beneficial schedules from the scheduling space. For example, if data is stored x-major in memory, then it makes sense to enforce loop `xi` of code 5.2 to be inner to `yi` and `ii`. The compute level should then also be kept above `xi`, to ensure a chunk with size T_x will be scheduled for transfer. By limiting T_x to multiplies of the burst size, it can be ensured that a block of continuous data is accessed every time.

Vectorisation

Targeting hardware with vectorisation capabilities is identical to targeting burst transfers. By selecting a loop for vectorisation, and limiting the tile size to multiples of the vector width, only beneficial schedules will be selected. Note that, depending on the capabilities of the target hardware, vectorisation can be orthogonal to optimising for burst transfers. For burst transfers the data layout in memory matters, while this restriction may not matter for vectorisation, or a rearrangement of data upon load may be cheap. As such T_x could for example be limited to match with a memory block size, while T_y is vectorised. This way multiple hardware features can potentially be optimised orthogonally. Note that

if this is not possible, it does not mean both can not be addressed. On most architectures one would expect that it is feasible to select a T_x that matches both with the burst size and the vectorisation support.

Instruction Memory Pressure

One aspect that is not accounted for in the presented models is control flow complexity when several layers are fused. With each layer that is added to a fused segment, the loop nest becomes deeper. This is evident when observing first code 5.2, which represents a single layer with tiling. After loop fusion, this code expands as show in code 5.3. For each layer, another six loops are added to the code. This inevitably increases the code size, which will put pressure on the instruction memory. In particular the outer loops, who's executions are relatively far apart, will start to suffer from misses in any instruction cache. However, based on the instruction cache size and inspection into the code size of the generated loop nests, it may be possible to estimate which loop levels will be pushed out of the instruction cache. For the iterations of the levels that do not fit, a penalty can be added to the execution time based on the target platform's instruction cache hierarchy.

Multi-level Memories

As the energy evaluation in section 5.7 demonstrates, the benefits of fusion are limited by the simple two-level memory hierarchy, i.e., one internal scratchpad and one external main memory, captured in the models. In contrast to burst transfers and vectorisation, handling multi-level memory is not a matter of limiting the design space. However, it is possible to extend the models with multiple levels of tiling, and multiple store/compute levels. If the target memory system is hierarchical, i.e., each level progressively contains a subset of the data, then the models can be updated to contain an explicit copy action. Essentially each layer can be prefixed with a number of 'dummy' layers, which represent the data of a layer in each memory level. The production of such a layer is simply defined as a copy from the previous layer. In the memory hierarchy this copy represents a transfer, as such these dummy layers will be referred to as transfer layers. The models can use the available attributes for layer fusion to fuse all transfer layers into the layer they belong to. The production of these layers, which represents a data transfer from one level to another one, can then be taken into account in the model for different tile sizes. Of course it is also

possible to skip a transfer layer altogether when layer fusion is used, such that intermediate data does not need to go through the entire memory system.

Throughput & Latency Models

The presented models only address operation and access counts, but not performance in terms of throughput or latency. Modelling these terms inherently requires platform knowledge, which makes it difficult if not impossible to derive a generic approach. However, given a specific platform, it may be possible to extend the presented models to include throughput and or latency.

When modelling throughput, it is important to determine whether a solution is compute bound, or memory bound. First both the compute and memory access models have to be scaled from raw counts to cycle counts. In the case of compute, this includes accounting for vectorisation, or other parallelisation techniques, to arrive at an operations per cycle model assuming a compute bound system. For the memory accesses, an analysis based on the target platform's memory hierarchy needs to be done in order to translate this to a bandwidth measure. Burst transfers can be taken into account here as described above. Once these two models have been obtained, roofline analysis can be performed to determine whether for a particular schedule the given platform is compute or memory bound. Based on the limiting factor, the system throughput can be estimated. If the target platform does not employ efficient prefetching, accurately estimating the bursty nature of memory traffic could be challenging. However, since the execution schedule is known at compile time, software prefetching techniques may be employed to spread memory accesses more evenly, countering this effect while simultaneously improving throughput.

Latency estimates may follow a similar approach. Based on access times of memory hierarchies at the target platform, a latency model can be derived. Again the (bursty) nature of memory access will be a determining factor in the accuracy of such a modelling. If (software) prefetching is applied effectively, it may be possible to completely hide the latency of all memory accesses with the exception of the very first couple of accesses. Since the schedule is statically known, prefetching should be very effective, but it should be noted that it will also come at a cost of increased memory footprint. The memory footprint models could be adapted to account for a certain window in time ahead, to model this loss accurately. In particular when software prefetching is applied, the scheduler will have full control of the amount of memory that should be reserved for prefetches.

5.8.3 Schedule Space Limitations

Although the presented models cover a vast design space, several limitations apply. In particular, only regular convolutional layers with striding are considered since most layers are of this type. However the models could be extended to cover different convolutional layers types, such as dense and depthwise convolution, and recurrent neural network layers. Dense layers can be modelled as a layer where the kernel spans the entire input, i.e., $\mathsf{D}_\mathsf{m} = \mathsf{D}_\mathsf{x}$ and $\mathsf{D}_\mathsf{n} = \mathsf{D}_\mathsf{y}$. For such dense layers it could pay off to also consider tiling of these kernels. Depthwise layers can be modelled by setting the number of input and output feature maps to one, i.e., $\mathsf{D}_\mathsf{i} = \mathsf{D}_\mathsf{z} = 1$, and multiplying the estimates by the number of original feature maps to compensate. Recurrent layers are slightly more complicated, since they use neuron states computed in a previous evaluation. This state could either be stored entirely in on-chip memory, or it could be transferred back and forth to external memory. In the latter case, it is probably possible to partly reuse the models for loading input data at the first layer in a fused section, and add them to the cost of an intermediate recurrent layer. The output state of such a recurrent layer always has to be transferred out, which potentially could be modelled by applying the output model to an intermediate recurrent layer.

Furthermore, fusion over skip, or residual, connections such as present in ResNet for example are not supported by the developed open source tool. Support could be provided by adding a binary choice to the network schedule per residual connection whether it should be stored in external or internal memory, analogous to the work by Goetschalckx et al. [40].

Finally one optimisation not covered by the presented models is the option to keep part of a layer in internal memory while transitioning from one fused segment to the next. This technique is included in the work of Goetschalckx et al. [40] as part of an optimistic model for schedules without fusion, but could be applied generically between fused segments as well. However, the resulting control code is likely quite complex, as the iteration order of consecutive segments needs to be reversed between layer transitions. For practical applications of this optimisation more research is required.

5.9 Conclusions

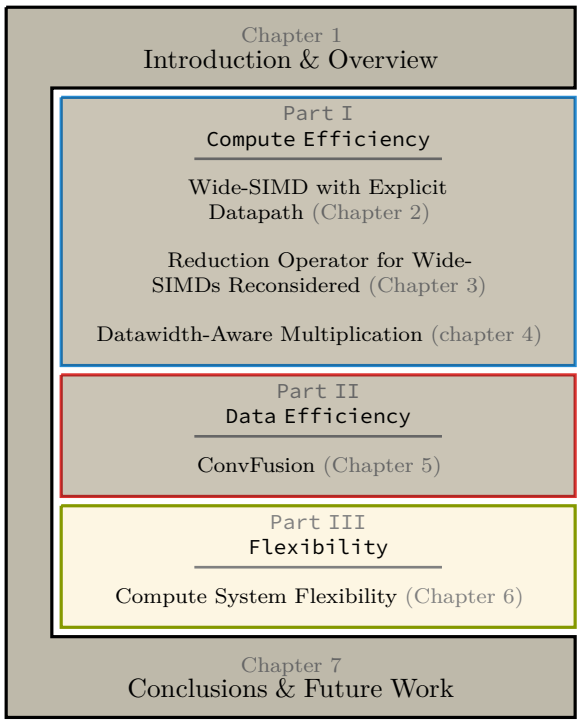
A practical scheduling space of convolution layers in CNNs has been outlined in section 5.3, including loop reordering, tiling, recomputation, and fusion.

Generic models on this design space have been proposed in section 5.4 for required external memory accesses, internal buffer space, and MACs. An efficient schedule space traversal method has been described in section 5.5, and an embodiment of the proposed models and schedule space traversal method has been described and published as open source tool [154]. Using this tool the accuracy of the proposed models has been verified on synthetic networks using instrumented Halide code [124]. The effects of various scheduling techniques, i.e., loop reordering and tiling, layer fusion, and recomputation, have been evaluated on six real world neural networks in section 5.6. An evaluation of the impact on energy consumption of these techniques has been provided in section 5.7.

Using code generation and instrumentation through Halide, the models have been verified against measurements and found to be accurate. By covering both layer fusion and tiling of weights, the proposed models have furthermore been shown to provide additional Pareto points compared the state of the art. This effect is most notable in networks which are weight dominated.

To capitalize on the benefits of fused scheduling techniques multi-level memory appears to be required. Section 5.8 discusses how the presented models may be extended to accurately model multi-level memory. The models presented in this chapter are hardware agnostic, and suggestions have been made in section 5.8 on how to limit the scheduling space to target real hardware that supports burst-accesses and vectorisation, increasing the applicability of the presented models.

Part III
Flexibility



Chapter 6

Compute System Flexibility

This chapter is based on the work published in "How Flexible is Your Computing System?" [66].

In literature, computer architectures are frequently claimed to be *highly flexible*, typically implying the existence of trade-offs between flexibility and performance or energy efficiency. Processor flexibility, however, is not very sharply defined, and consequently these claims can not be validated, nor can such hypothetical relations be fully understood and exploited in the design of computing systems. This chapter is an attempt to introduce scientific rigour to the notion of flexibility in computing systems. A survey is conducted to provide an overview of references to flexibility in literature, both in the computer architecture domain as well as related fields. A classification is introduced to categorize different views on flexibility, which ultimately form the foundation for a qualitative definition of flexibility. Departing from the qualitative definition of flexibility, a generic quantifiable metric is proposed, enabling valid quantitative comparison of the flexibility of various architectures. To validate the proposed method, and evaluate the relation between the proposed metric and the general notion of flexibility, the flexibility metric is measured for 25 computing systems, including central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs), and 40 application-specific instruction-set processors (ASIPs) taken from literature. The obtained results provide insights into some of the speculative trade-offs between flexibility and properties such as energy efficiency and area efficiency.

6.1 Introduction

Arguably one of the most famous books in the field is "Computer Architecture — A Quantitative approach" by John L. Hennessy and David A. Patterson [61]. The title itself is concise and apt, so it is interesting the authors opted to add this particular subtitle: a *quantitative* approach. It implies the belief that

quantifying design choices ultimately leads to better computer architectures, a message that certainly could be directed towards those who make claims about *flexible* architectures without means of quantifying these claims, or even without as much as a commonly accepted qualitative definition of flexibility. With Moore's law seemingly coming to an end, new advancements in computing will have to be made on the architectural side. To advance the state of the art, fundamental understanding of various trade-offs in computer design is vital. The way forward, therefore, is a quantitative one.

Many key system properties such as performance, power dissipation, and energy efficiency are all well defined in a quantitative manner. With these metrics in place, quantitative, objective comparisons can be conducted between different machines. For flexibility however, such a quantitative (and even qualitative!) definition is lacking, despite its increasing importance in system design. In product research and development, computing platforms are required to sufficiently support new or updated algorithms, as algorithms are changing at a striking speed. Exemplary are the current developments in artificial intelligence, which result in new compute-intensive algorithms at a high cadence. Such rapidly developing markets require systems that can deal with changing applications, which is the property flexibility typically seems to refer to. However, in absence of a proper definition, it is impossible to make solid statements, and compare designs on flexibility.

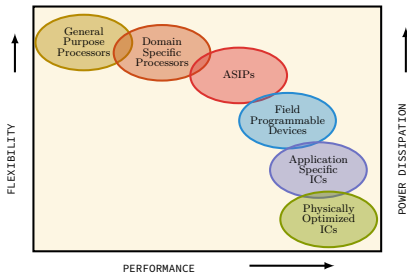
Despite the lack of a formal definition of flexibility, there appear to be some commonly accepted notions surrounding flexibility. In particular, flexibility seems mainly used to refer to the adaptability of processors to different applications. This leads to the common idea that a programmable processor which can be reused across applications is 'flexible'. On the other hand, a processor with fixed logic such as an ASIC cannot adapt, exposing its inflexibility [106]. As can be seen in figure 6.1 the authors of these figures appear to agree with this sentiment. However there are also some contradictions to this view on flexibility. For example in figure 6.1a, among programmable processors the field-programmable devices are claimed by the authors to be less flexible than software programmable processors due to their inadequate programmability [76]. Unfortunately the term "programmability" is also ill-defined here. Perhaps the best definition of programmability in existence is to check Turing completeness of a programmable device, but this would leave only two classes of programmability making it a measure with low practical value. Another perspective on flexibility refers to how well a processor supports different applications, in which case field-programmable gate arrays (FPGAs) could be seen as the most flexible, since

any hardware, including digital signal processors (DSPs), graphics processing units (GPUs), and central processing units (CPUs), can be instantiated on FPGAs. Apart from this debate on how to rank the flexibility of architecture classes, perhaps even more worrisome are the contradicting claims on relations between flexibility and other metrics. In figure 6.1d A. Osman El-Rayis equates flexibility to area, whereas Tobias Noll sees it as directly related to power dissipation in figure 6.1a. While this is definitely not an exhaustive list of views on flexibility, it painfully exposes how the lack of a formal definition leads to a wild-west of claims and conflicting visions, none of which can be backed up with objective measurements.

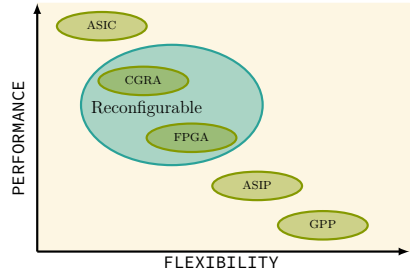
Despite the greatly varying interpretations of flexibility, many seem to agree that there may be interesting relations and trade-offs between flexibility and other properties, such as performance and energy efficiency. As illustrated in figure 6.1, processing architectures alternatives have been evaluated and ranked in terms of flexibility, performance, power dissipation, and area [75, 76, 105, 171, 126]. A variety of architectures have been developed which claim to balance energy efficiency and flexibility. The development of domain-specific functional units and a transition to heterogeneous multi-core systems are a testament to this notion [83, 49, 32]. These hypothetical relations suggest that understanding flexibility is beneficial when designing a system, enabling informed trade-offs.

To overcome the lack of understanding of flexibility, this chapter sets out to provide both a qualitative and quantitative definition of flexibility. It should be noted though that, with such a fragmented landscape of interpretations of flexibility, the authors are under no illusion that it is possible to unify the field and reach consensus without a wider discussion. Instead, this chapter is to be seen as a first attempt, which does not so much aspire to provide a definitive answer, as it hopes to be thought provoking and spark a discussion within the community.

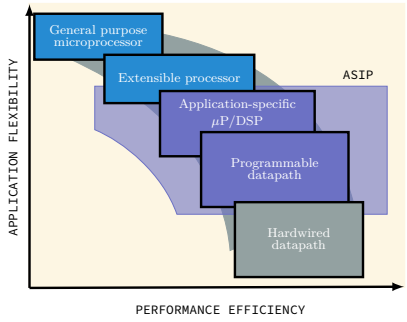
To arrive at a quantitative measure for flexibility, a qualitative definition is established by exploring uses of the term flexibility in literature and then examining various options. Based on this qualitative definition, a quantitative measure is derived. In the translation of flexibility, from a qualitative term to a quantitative definition, there exist several degrees of freedom. The final choices made in this translation are motivated extensively. However, more importantly, the alternatives are discussed systematically in similar detail. The intention is that this systematic approach can provide an initial framework for a broader



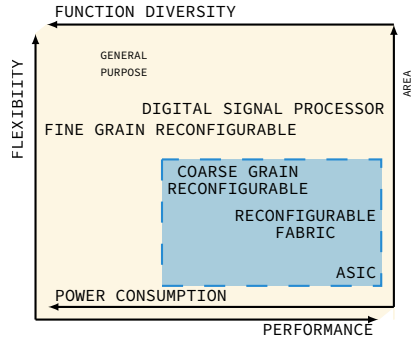
(a) Flexibility related to performance and power according to T. Noll [75, 76]. Note that according to this figure flexibility is directly related to power dissipation.



(b) Performance versus flexibility plot by G. Ndu [105]. The ordering of the flexibility of architecture classes aligns quite well with the ordering given in figure 6.1a, although the curve is quite different.



(c) 'Application Flexibility' versus 'Performance Efficiency' by M. Willems from Synopsys [171]. Unfortunately no further definition of both these metrics is provided in the accompanying text.



(d) An interesting graph by A. Osman El-Rayis [126] which relates many metrics including flexibility. In contrast to figure 6.1a flexibility is here claimed to be directly related to area.

Figure 6.1: Collection of published figures with claims about the flexibility of architecture classes, and relations between flexibility and other metrics such as performance and power. Note that none of the axis in these figures are labelled with units.

discussion in the community on how flexibility should be defined, such that eventually a standard accepted metric can be established.

To validate the metric proposed in this chapter, in total 14 applications are benchmarked on 25 different commercial of the shelf (COTS) platforms. It is shown that results align with several common concepts of flexibility found in literature. For example, GPUs deliver the highest performance in general for the used parallel benchmarks, but sacrifice in terms of flexibility, compared to general purpose processors (GPPs) in figure 6.1a. Furthermore the flexibility of 40 application-specific instruction-set processors (ASIPs) from literature is determined to evaluate the relation between specialisation [33] and flexibility.

The remainder of this chapter is organised as follows: section 6.2 presents a survey on flexibility definitions in the literature, both in the field of computer architecture, as well as related technical fields. Section 6.3 introduces both a qualitative and quantitative definition for processor flexibility based on the collected views in the survey. A novel normalization method based on the intrinsic workload of applications is included in section 6.4. Section 6.5 explains the experimental setup, the implementation of the workload estimator, and the methodologies applied in this chapter. The flexibility results are analysed in section 6.6. Comparison with alternative definitions is provided in section 6.7. In-depth reflection and extensive discussion are presented in section 6.8, which places the proposed definition in context of the field. Finally, section 6.9 concludes this chapter.

6.2 Survey of Flexibility in Literature

In the field of computer architecture few studies have striven to define and quantify processor flexibility. Therefore, this section starts with discussing the existing flexibility definitions in other fields.

Various other fields have more properly defined flexibility, as demonstrated by the three following examples:

- A generic viewpoint on system flexibility is provided by Chryssolouris, who defines flexibility as the sensitivity of a system to (external) *changes* [24]. Lower sensitivity is understood to indicate higher flexibility, as the system is apparently able to operate relatively unaffectedly under the external changes.

- Conceptually identical is the definition proposed by Kellerer et al. who state that the flexibility of electronic networks refers to the ability to support new requests, such as *changes* in the requirements or new traffic distributions [78].
- In power systems, flexibility is also based on external *changes*. More precisely it is defined as the ability of a power system to deploy its resources in response to *changes* in the net load, which is the residual demand that must be supplied after the depletion of renewable energy [88].

In general these examples consider flexibility as a system property and quantify flexibility as the *insensitivity of the system based on external changes*, instead of formulating flexibility as a function of diverse system parameters. This approach is transferable to computing systems, as will be outlined in this chapter.

In literature related to computer architecture, approaches to define processor flexibility can be divided in two categories:

1. Definitions that regard flexibility as an *intrinsic static property* of a system.
2. Definitions that regard flexibility as an *extrinsic mutable property* of a system, dependent on and measured under the influence of external applications.

Works that fall into the first category are discussed in section 6.2.1, while the second category is elaborated in section 6.2.2.

6.2.1 Definitions of flexibility as an intrinsic static property

One definition that regards flexibility as an intrinsic property is proposed by Stigall et al. [143] as early as 1975. In their definition a computer is seen as major memory/compute units and their datapath connections. The more connections between the major components, the more options the machine has, and thus the more flexible it is. The authors also quantify flexibility, namely as the ratio of the number of data paths that connect major components to the maximum possible number. The idea is interesting but does not seem to hold for modern machines. For example, when directly applying this concept to an single instruction multiple data (SIMD) machine and a multi-core with the same number of cores, an SIMD processor that has fewer components (only a single instruction memory and decoder, instead of one per core), would usually have higher connectivity among components than a multi-core processor, implying higher flexibility. This conclusion seems counter-intuitive however, since an SIMD

can only execute a single instruction at a time on all compute elements, while a multicore can execute different instructions on its compute units. Therefore the set of operation modes of the multi-core is a superset of the SIMD modes, which implies an SIMD can not be more flexible than a multi-core system.

Another intrinsic definition category is processor versatility as proposed by K. van Berkel [77]. As shown in equation (6.1), processor versatility is defined as the average number of instruction bits per useful operation. The amount of useful operations is extracted according to a complexity analysis of a single algorithm. Based on the intuition that when more bits are used to encode instructions, the processor is more versatile as more options are available. Therefore versatility is a property of the instruction set architecture (ISA) and is independent of particular implementations or executed applications. For instance, versatility increases if the ISA is extended with special instructions serving dedicated hardware, while potentially being useless to accelerate the target applications. Although the work by K. van Berkel [77] is the most rigorous attempt to formally define flexibility to date, some aspects hinder its application. Conducting complexity analysis of applications to obtain useful operations is challenging without manual effort. Using operations as basic units implies that different operations are weighted equally, such as multiplication and addition. This seems somewhat arbitrary given that the area or energy footprint of a hardware multiplier for example is many times that of an adder in the same technology. Another weakness of this metric is that it can not be applied to all systems, such as processors that do not execute clock-based instructions, e.g., FPGAs.

$$versatility = \frac{average\ instruction\ size}{number\ of\ useful\ operations\ per\ instruction} \tag{6.1}$$

However, the concept of measuring the required bits to execute a task is intriguing, and possibly has a use of its own. Therefore this versatility metric is further discussed in section 6.7.1.

6.2.2 Definitions of flexibility as an extrinsic mutable property

Most flexibility definitions fall into the second, extrinsic category, and are considered a property measured under the influence of external applications. Sze et al., for example, provide the following view on flexibility: *"Flexibility refers to the range of deep neural network (DNN) models that can be supported on the DNN processor and the ability of the software environment (e.g., the*

mapper) to maximally exploit the capabilities of the hardware for any desired DNN model." [145]. Here the DNN models, i.e., extrinsic properties, are used to define the flexibility of a system. Their conclusion on flexibility therefore also is: *"... to assess the flexibility of DNN processors, its efficiency (e.g., inferences per second, inferences per joule) should be evaluated on a wide range of DNN models"*. Although not a quantifiable definition per se, the notion of defining the flexibility of a system in relation to the influence of relevant extrinsic properties is clearly present. Furthermore it is important to note that various forms of flexibility are implied, since the effect of an extrinsic property can be expressed in terms of performance (inferences per second), and energy efficiency (inferences per joule). Also interesting is the inclusion of the software (mapper) into the equation, i.e., system flexibility does not only depend on the hardware, but also on the supporting software.

Tomusk et al. propose to quantify the flexibility of Single-ISA heterogeneous processors with entropy-based diversity [150]. The idea is that different cores in a flexible heterogeneous processor can cover more of the design space. Exploring the design space is achieved by selecting the cores of the system that are Pareto optimal for power and performance. Higher spread on these Pareto cores means better flexibility. However, this definition is strictly limited to heterogeneous processors, ruling it out as a general definition for all computing systems.

Fasthuber et al. propose a different model to define computer architecture flexibility [32]. The proposed model extracts system requirements from a set of applications to check if architectures provide sufficient flexibility to support the minimum requirements in a true/false manner. For instance, in case a hardware divider is imperative to reach the required performance for a division, a processor performing division by software emulation fails to meet the performance requirement. This model assesses architecture flexibility based on external applications, examining how well architectures support diverse applications. However, it is challenging to apply this model generically because of the need for a set of hard boolean requirements. Moreover, the range of the scale is severely limited, since the flexibility is the result of counting the number of met requirements. If the set of requirements is small, distinguishing various systems may be impossible.

Apart from the intrinsic versatility metric as proposed by Van Berkel [77], there is a competing extrinsic definition for versatility by Rabbah et al. [123]. To distinguish the two versatilities in this thesis, the metric defined by Rabbah et al. in their VersaBench paper will be referred to as "VersaBench Versatility" or V_s.

VersaBench Versatility is defined as the geometric mean (GM) of a processor's performance over a vector $\boldsymbol{X} = [x_1 \cdots x_n]$ of benchmark applications, normalised to the best execution time known for each of those applications $t_{\text{fastest}}(x_i)$, where $x_i \in \boldsymbol{X}$ (equation (6.2)).

$$V_s = \left(\prod_{i=1}^{n} \frac{t_{\text{fastest}}(x_i)}{t_{\text{s}}(x_i)} \right)^{\frac{1}{n}} \tag{6.2}$$

This can be interpreted as the mean slowdown of a processor compared to an idealized fastest known execution time per application. As such VersaBench Versatility has a range of $0 < V_s \leq 1$. Compared to the definitions provided by Tomusk et al. and Fasthuber et al., this approach resolves the limitations on applicability of the metric to more diverse architectures. The use of only execution time measurements further increases the practicality of the proposed metric. However, improvements in VersaBench Versatility can result from an absolute increase in performance, i.e., an increase in clock speed to boost performance can effectively improve VersaBench Versatility. This makes VersaBench versatility and performance directly related, which we argue should be distinct, orthogonal features. In-depth analysis and comparison with the flexibility metric as proposed later in this chapter are provided in section 6.7.2, and furthermore show that the normalization proposed by Rabbah et al. [123] is a mathematical unit operation and does not contribute to a change in the final V_s.

Although not explicitly targeting flexibility, the work of Fisher et al. [33] on customizing processors exhibits interesting parallels with work on flexibility in literature. In their work Fisher et al. set out to optimize a VLIW processor for a set of tasks, which is again the extrinsic factor. It is argued that instead of optimizing a processor for only one application, some performance may be sacrificed for that application to achieve a better average performance for the entire dataset. In essence, by sacrificing performance for one application to benefit the overall benchmark set, it can be argued that the flexibility of the processor has been improved. It is important to distinguish though that the method applied by Fisher et al. still tries to obtain a higher overall performance, and does not necessarily consider minimizing the impact of external changes. A quantified analysis of this method and how it relates to the concept of flexibility is provided in section 6.6.2.

The related work discussed in this section shows that there are tremendous diversities in understanding and quantifying flexibility. Overall, flexibility has more frequently been defined as an extrinsic metric based on *external changes*,

than an intrinsic property. However, it can also be observed that the inherent properties of most of those definitions limit the scope of application. With this in mind, the metric proposed in this chapter aims to avoid this pitfall, and also be practical to apply generically.

6.3 Defining Flexibility

Despite a few valiant attempts to define flexibility for computing systems in the existing literature, it can be concluded there is no consensus in the community as to what flexibility exactly is, let alone how to objectively measure it. Unifying the various views on the topic into a single coherent definition is a daunting task, yet one that has to be faced if the rewards are to be reaped. This section outlines our attempt at defining flexibility for computing systems. Starting from a qualitative definition in section 6.3.1, a quantitative metric is then derived in section 6.3.2. In section 6.3.2, several crucial properties of a universal flexibility metric are defined and proven to hold for the proposed metric. Finally section 6.3.3 details the scope of applying the proposed metric.

6.3.1 Qualitative Definition

Before determining a quantitative definition of flexibility, there has to be agreement on a qualitative definition. As outlined in section 6.2, a recurring theme when dealing with flexibility in literature is *external changes*, or in particular, a system's *response* to external changes. A natural translation of this notion to computing systems is to regard changing applications as the external changes, while any secondary metric, such as performance or energy efficiency, can be used to express a system's response. Consider for example the benchmark data for two systems in figure 6.2. As applications change, so does the (normalised) performance of these systems.

For the particular case in figure 6.2: *Which system is more flexible?*, Some may argue System `I` is more flexible, as System `I` can maintain the highest average performance when applications change. However, we postulate such reasoning is a fallacy, and that performance has to be an orthogonal measure to flexibility. Were this not the case, then flexibility would merely be a synonym of "average performance", and not an independent metric as appears to be the common notion. Rather, we argue that the system which supports different applications equally well is more flexible, regardless of its average performance. In figure 6.2, System `II` obviously has lower performance than System `I`, however,

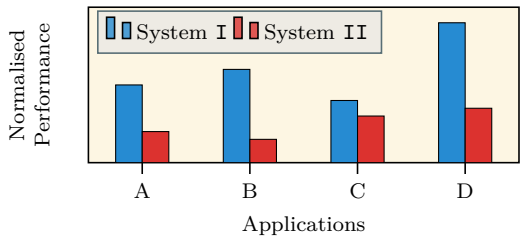


Figure 6.2: Given the performance of two hypothetical systems I and II, normalised to some baseline system. System I consistently outperforms System II. However, the performance of System II is much less influenced by changing applications. Which system is more flexible?

it is stabler under application changes. This is a desirable property orthogonal to performance. For example, during the design of a computing platform with cost constraints, a processor has to be selected but the final applications are still subject to change. In this case, it could be beneficial to select a processor with overall lower, yet sufficient, performance, but higher flexibility, such that the performance is likely to be still sufficient when applications do change.

In the case of figure 6.2 flexibility is defined in relation to performance variability. However, other established metrics can be used freely, such as energy efficiency, area efficiency, or a hybrid cost function. How much the secondary metric changes resulting from changes in applications is an indication of the flexibility of the platform in that regard.

Based on the reasoning above, we arrive at the following qualitative definition of flexibility:

> *Compute system flexibility refers to the invariance of a system's normalised*[1] *performance, energy efficiency, area efficiency (or other secondary metrics), to change of application.*

In particular, when the secondary metric is affected more by changes in an application, the system is considered to be less flexible.

Although it is just a qualitative definition, some general observations as to how it aligns with several notions regarding flexibility can already be made. Consider, for example, an arbitrary set of benchmark applications that expose different levels of data-level parallelism (data-level parallelism (DLP)). When mapped to

[1] N.B.: This 'normalization' is further clarified in section 6.3.2

a GPU, applications with high levels of DLP would benefit from the many vector cores and achieve high performance. Applications with limited DLP, however, would not be able to run efficiently on a GPU, and consequently, achieve low performance in comparison. Thus, for a mixed, arbitrary benchmark set a GPU would not be very flexible. In contrast, a simple single-core CPU without vector extension would not provide an unbalanced advantage for applications with high levels of DLP. Therefore, it would be ranked more flexible than the GPU, which aligns with commonly accepted notions. Similar examples can be made for various classes of architectures, such as CPUs with advanced branch predictions and applications with complex control flow, or DSPs and algorithms that require multiple floating-point multiply accumulate (MAC) operations. Specialization towards only a subset of relevant applications may improve overall performance but could degrade flexibility.

The preceding example also illustrates that the selection of benchmark applications is an important factor in determining a system's flexibility. After all, if a dedicated parallel benchmark set was selected, the GPU would be ranked as more flexible. It is worth pointing out this is not a weakness nor flaw in the definition of flexibility, but merely emphasizes that benchmarks should be selected based on the application domain a system is targeting. Similarly, it makes no sense to use a graphics benchmark on a CPU, when the system is targeted for handling search engine queries. Proper benchmark set selection is just as crucial to obtain meaningful flexibility results as it is for measuring any other system property. When done properly though, the obtained flexibility ranking will be representative for the selected application domains.

Finally, it is highly important to note that flexibility as defined here can be artificially raised. By inserting `nop` operations in the fastest applications, the performance of all applications can be lowered to match the lowest-performing application. This would result in the most flexible system, although the overall achieved performance is degraded. For real-time systems, such an approach may not even be undesirable, as long as the performance requirements are still met. However, to account for the loss in performance, energy efficiency, or any other secondary metric, flexibility should always be reported coupled to these metrics. One possible way to couple flexibility with other metrics is through a compound metric, such as the classical energy-delay product (ED), or energy-delay-power product (EDP).

6.3.2 Quantitative Definition

This section translates the qualitative definition of flexibility to a quantitative measure. In particular, the focus is on how to quantify "the invariance of a system's response". A measure has to be found which expresses variations in system performance, energy efficiency, or other secondary metrics. Several such measures for quantifying statistical variation among data points exist. This section qualitatively explores these options, and finally selects the most suited approach to quantify flexibility.

There are two classes of variation measures:

1. *Robust measures* are resilient to extreme values in a dataset, and in general, try to reduce the effect of outliers in the data. A typical example of robust measures is the median absolute deviation (MAD) (equation (6.3)), defined as the median of the absolute deviations from the median of the original data. The MAD ignores a small number of extreme values, and only focuses on the median of the dataset.

$$MAD(\boldsymbol{X}) = median\left(|x_i - median(\boldsymbol{X})|\right), \tag{6.3}$$

   where $\boldsymbol{X} = [x_1 \cdots x_n]$ is a vector of measurements.

2. *Conventional measures*, in contrast, are sensitive to extreme values [170]. Arithmetic standard deviation (ASD) (equation (6.4)) and geometric standard deviation (GSD) (equation (6.5)) are two such conventional measures, both of which describe the dispersion degree of a data set.

$$ASD(\boldsymbol{X}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(x_i - AM(\boldsymbol{X})\right)^2}, \tag{6.4}$$

   where $AM(\boldsymbol{X}) = \frac{1}{n}\sum_{i=1}^{n} x_i$ is the arithmetic mean.

$$GSD(\boldsymbol{X}) = \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\boldsymbol{X})}\right)^2}\right), \tag{6.5}$$

   where $GM(\boldsymbol{X}) = \left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}}$ is the geometric mean.

Robust measures are particularly suited for noisy measurements with plenty of data points. In computer architecture, however, measurements are easily repeatable, allowing noise to be filtered by alternative means. Furthermore, the number

of applications in benchmark sets is often quite limited, and as such ignoring points risks ignoring important data. Therefore the conventional measures are more suited to represent dispersity in a set of benchmark applications.

The fundamental difference between the two conventional measures is the used average: the arithmetic mean (AM) versus the geometric mean (GM). The AM simply characterizes the average value of the dataset by dividing the sum of all points by the length of the dataset. Thus, the ASD indicates the average distance of data points in the dataset to the AM, and has the same unit as the dataset. Since the AM and ASD are sum-based values, they are appropriate for additive processes. Different from the AM, the GM takes the product of all numbers, and then raises it to the inverse of the length of the dataset. Because of this, the GSD as defined in equation (6.5) is a multiplicative factor and does not maintain the original dimension of the data [79]. When dealing with multiplicative relationships such as growth rate and speedup, the ASD over-estimates data dispersity, while the GSD as a product-based value is the correct average to use [34, 104].

Based on this, the GSD is selected as the measure of dispersity for flexibility. In particular because, as also stated in the qualitative flexibility definition, benchmark data is to be first normalised when deriving flexibility. In the case of performance, the inverse of absolute runtime would not give an accurate view of which applications are supported better than others. Some applications may simply require more work than others, and thus the runtime needs to be normalised (more on this normalization in section 6.4). Similarly, energy efficiency is the energy consumption normalised to the amount of work performed by each application. This normalization results in a multiplicative relation to the normalization baseline, and thus GSD is the only correct measure of dispersity.

Concluding the quantitative definition of flexibility:

> *Compute system flexibility is defined as the inverse of the geometric standard deviation of a system's normalised performance, energy efficiency, or other secondary metric, within a benchmark set with measurement vector* $\boldsymbol{X} = [x_1, \cdots, x_n]$ *(equation* (6.6)*).*

$$Flexibility(\boldsymbol{X}) = [GSD(\boldsymbol{X})]^{-1} = \exp\left(-\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\boldsymbol{X})}\right)^2}\right), \quad (6.6)$$

where $GM(\boldsymbol{X}) = \left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}}$ is the geometric mean.

Flexibility Metric Properties

Validation of a new metric is a paradox, as there is no established ground truth available. However, it is possible to derive several necessary properties a flexibility metric should adhere to. This section discusses these properties and proves them for the proposed metric.

1. Flexibility in performance should be independent of scaling the platform frequency (equally across all benchmarks). I.e., if a given platform executes applications at F cycles per second, it should measure the same flexibility if it for example runs at $0.5F$ for all applications. This holds for the proposed metric as the GSD is invariant to multiplicative scaling (see lemma 3). Note that this scaling over all benchmarks also holds for platforms that have a difference frequency per application, such as FPGAs. If all applications are executed at half *their* original speed, the measured flexibility will be the same.

2. Stricter than the previous property, flexibility should be independent of performance. Indicated by lemma 1 and lemma 2, the increase of an element in a positive dataset always results in an increasing GM. However, the GSD value can increase or decrease, which depends on how the increase of an element impacts dataset diversity.

The two properties described are absolutely essential for any flexibility metric. Apart from these two, there are two properties which are nice to have, but not strictly required. For completeness it is shown that these property hold for the proposed metric.

3. Preferably, flexibility, as a multiplicative measure, is invariant to using the reciprocal of the underlying metric. This property is particularly useful as it decouples clock frequency, and with that to some degree the technology node, from flexibility. Furthermore, in the case of flexibility in relation to energy, the flexibility calculated over $^{\text{J}}/_{op}$ will be the same as the flexibility calculated over $^{op}/_{\text{J}}$. Lemma lemma 7 shows that this is the case for the proposed definition, since $GSD\left(^{\boldsymbol{X}}/_{\boldsymbol{Y}}\right) = GSD\left(^{\boldsymbol{Y}}/_{\boldsymbol{X}}\right)$.

4. It is convenient if the metric is bound to a fixed range. In the case of the proposed metric it can trivially be shown that flexibility has a range of $(0, 1]$. A flexibility of one is achieved when an architecture has the exact same speedup for each application in the benchmark set compared to the normalization reference, as further explained in section 6.4.

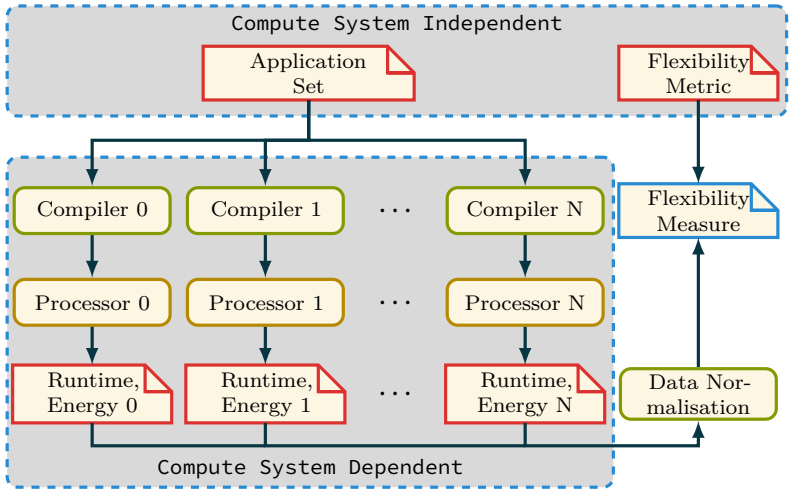


Figure 6.3: Framework for flexibility measurements. Note that the compilers are in the chain of measurement, and are as such considered part of the total compute system.

6.3.3 Flexibility Scope

One aspect of flexibility that has not been addressed so far is the measurement scope. In the case of performance measurements, it is not only the performance of a processor that is measured, but also that of the surrounding memory system, interfaces, and even the compiler. This is not exclusive to performance measurements but also holds for energy, power, and many other metrics. The fact that the compiler is part of the measured system is in fact common practice, but often overlooked when publishing results. Therefore we like to explicitly state the compiler used to perform benchmarking of a system should always be part of the results.

For the flexibility measurements in this chapter, the system border is drawn at the compiler, and the benchmark code itself is taken as universal for all platforms. When measuring across different platforms this may not always be feasible, so different choices can be made in specific situations. The recommendation is, however, to use a cross-platform language benchmark such as OpenCL. A benchmark set with multi-language support is a good alternative if no single language can support the systems under test. In general, the procedure of measuring flexibility will follow the flow illustrated in figure 6.3.

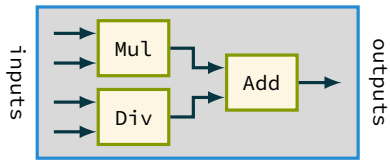


Figure 6.4: Application expressed as a single combinatorial circuit.

The first step is to compile and run the same application set on the target systems, and measure the desired secondary metrics such as performance and energy consumption per benchmark. Next follows normalization of the obtained results. Data normalization is a prerequisite to ensure benchmark results of diverse applications are comparable, as further discussed in section 6.4. Finally, the proposed flexibility metric is computed from the normalised data, resulting in a flexibility ranking of the measured platforms.

6.4 Normalization to Intrinsic Work

Before the GSD can be computed, secondary metrics such as execution time and energy consumption, measured from diverse applications, need to be normalised. The reason is that typically applications in a benchmark set represent inequivalent computational workloads. For instance, applying Gaussian filters with different kernel sizes results in different workloads. Therefore, normalization based on workload is required before any data analysis and comparison [111].

The general approach is normalizing to a reference set, i.e., a each application x_i of a vector of applications $\boldsymbol{X} = [x_1, \cdots, x_n]$ is normalised according to:

$$m_{norm\_baseline}(x_i) = \frac{m_{target}(x_i)}{m_{baseline}(x_i)},$$

where m_{target} evaluates an arbitrary metric m for the target architecture for application x_i, and $m_{baseline}$ does the same for x_i on the baseline machine. However, determining a proper baseline for flexibility poses a challenge. Simply taking benchmark results of one system as reference implicitly makes the baseline system "the most flexible" by definition. For instance, when using a basic reduced instruction set computer (RISC)-type processor as the baseline as is often done by Hennessy and Patterson [61], normalizing to itself transforms each value in the dataset to one, resulting in no deviation and a flexibility of one. Consequently no

system could then be more flexible than the baseline RISC processor, or whichever platform is selected as the baseline. This choice seems rather arbitrary, and a more fundamental baseline is desirable.

The key concept is that normalization is applied to equalize an imbalance in workload that each application represents. The underlying notion is thus that applications describe a certain amount of *work*. Normalization on this *intrinsic workload* W_{int} yields the following normalization procedure:

$$m_{norm\_intrinsic\_workload}(x_i) = \frac{m_{target}(x_i)}{W_{int}(x_i)}$$

This normalization results in measures as "intrinsic work per second" for performance, and "energy per unit of intrinsic work" for energy efficiency, yielding comparable numbers between various platforms. Unfortunately, a measure for intrinsic workload also does not exist. There are many possible viewpoints on how intrinsic workload could be defined, yet this section will focus only on the one selected for normalization in this chapter. Section 6.8 will on the other hand explore several alternatives.

Under the assumption that computation indeed is equivalent to work (something that can be questioned from a physics point of view as will be discussed in section 6.8), the problem condenses to finding a unit for this work. Something often used in computer architecture is to count one RISC instruction as one unit of work. However, this implies multiplication and even division would represent the same amount of work as an addition or a logic and-operation. This is rather counter-intuitive, as the hardware complexity of a hardware divider is significantly greater than that of a logic and-operation, i.e., $\mathcal{O}(b^2)$ for a b-bit wide division, versus $\mathcal{O}(b)$ for a logic and of b bits. The implies division is fundamentally complexer than a logic operation. A possible way of weighting RISC instructions then is by the complexity of their equivalent combinatorial circuits.

Taking this one step further, the division of work into RISC operations is also rather arbitrary. From a purely theoretical viewpoint, it can be argued that the workload of an application is represented by its combinatorial circuit. I.e., the combinatorial circuit that statically represents the entire application, reading inputs and producing the final outputs without a (clocked) state in between as illustrated in figure 6.4. Such a circuit would clearly be completely impractical, but it can be argued than when written in a minimal form, i.e., with minimal basic gate (2 input — 1 output) count, it represents the intrinsic workload of the

application. The gates toggling during the execution of this circuit approximate the minimum required toggles to complete the computation. Interestingly memory and control flow operations are not required in such a completely combinatorial, spatial circuit, demonstrating that such operations are in essence an artefact of stateful Turing machines.

Note that this massive combinatorial circuit would have to be written in minimal form though, something intractable with modern technology since logic minimization is proven to be non-polynomial [19]. From a theoretical point, obtaining such a minimal circuit would be interesting, but to arrive at a practical measure for flexibility as is the goal of this chapter, a more pragmatic approach has to be employed. Rather than approaching this minimal circuit bottom up, one solution is to return to the roots of practical computing and approach it top down. Instead of finding the absolute minimal circuit, the circuit can be divided into elementary blocks with common functionality. Optimizations are not employed across these blocks to keep the design tractable. As the only requirement for these elementary blocks is that the set is Turing complete, there are many options. Here, however, it is possible to fall back on decades of research in computer design. A natural choice would be RISC-like elementary operations, such as addition and multiplication. Essentially, this circles back to the earlier idea of weighing RISC operations by the complexity of their equivalent circuitry, but with a notable exception: Those operations in an application that deal with control flow operations should not be counted towards the intrinsic workload. Since memory operations are also an artefact of stateful machines, they could too be omitted, but since they play an ever more important role in modern technology this chapter proposes several methods to still take memory into account. The effect of memory operations is further investigated in the experiments described in section 6.5.

Practically the intrinsic workload of an application as defined above can be approximated automatically by leveraging the intermediate representation (IR) of the LLVM compiler framework. To be able to deal with many input languages and target platforms, LLVM front-ends translate code into a generic intermediate instruction set, the IR. From this generic IR, the back-ends generate target-specific code. This IR has to be very generic to support as many languages and platforms as possible, and as such is a good candidate to use in an automated intrinsic workload estimator. Furthermore, it has the advantage that operations related to control flow and memory are distinguishable from other operations, and as such can be rejected for the workload estimation. This particular approximation gives rise to the following definition:

Approximated intrinsic workload is given by the dynamic IR instruction count of all operations not related to control flow (and optionally memory), weighted according to the circuit complexity of the operations.

More details on how this procedure is automated, and the weighting of the individual IR instructions used in this chapter can be found in section 6.5.4.

6.5 Experimental Setup

Now that a flexibility metric has been established, measurements of various systems can be performed to both validate the metric against commonly accepted ideas surrounding flexibility, as well as investigate hypothetical relations between performance, energy, area, and flexibility. To achieve this a wide spectrum of computer architectures is examined in this chapter, including COTS CPUs, GPUs, FPGAs, and DSPs. The selection 25 of these COTS systems, the selected benchmark set, and details regarding compiler settings and performed experiments are described in sections 6.5.1 to 6.5.3 respectively. The workload estimation of the benchmark set is captured in section 6.5.4. Section 6.5.5 describes the measured properties and their relations. Inspired by the work of Fisher et al. [33] the flexibility of custom, or ASIPs is also investigated. The setup and parameters of the related experiments are described in section 6.5.6.

6.5.1 Selected Systems

In this chapter flexibility measurements are conducted on 25 different systems. Applications are directly executed on real GPUs and CPUs. For the embedded DSPs, cycle-accurate simulators form the manufacturer have been employed to extract execution times. The FPGAs, finally, have been characterized using high-level synthesis (HLS) combined with post place & route clock speed reporting. Note that in contrast to the other considered systems, for the FPGAs the clock-speed in fact varies per application, as measured using post place & route clock speed estimation per application, target pair. Compilers, as a part of target systems, are tuned for maximum optimization where possible, to exploit the capability of the target processors as good as possible with the given code-base.

- **GPU:** Aimed at comprehending the difference of flexibility between desktop and embedded GPUs, one embedded GPU (Tegra K1) and three desktop GPUs are evaluated in this chapter by compiling the CUDA [109] versions

Table 6.1: Overview of used GPUs.

Processor	Chip	Archi	#Cores	Compiler
Tegra K1	GK20A	Kepler	192	nvcc 6.5
GTX 570	GF110	Fermi	480	nvcc 7.5
GTX TITAN	GK110	Kepler	2688	nvcc 7.0
GTX 750 TI	GM107	Maxwell	640	nvcc 7.0

Table 6.2: Overview of CPUs.

Processor	ISA	Micro-architecture	Cores	#Threads	Compiler
i7-6700	x86_64	Skylake	4	8	gcc 4.8
i7-4770	x86_64	Haswell	4	8	gcc 4.8
i7-960	x86_64	Bloomfield	4	8	gcc 4.8
i7-950	x86_64	Bloomfield	4	8	gcc 4.8
i7-920	x86_64	Bloomfield	4	8	gcc 4.8
Pentium 4	x86_64	Northwood	1	2	gcc 4.8
Processor	**ISA**	**System**	**Cores**	**#Threads**	**Compiler**
Cortex A15	ARMv7	Nvidia JTK1	4+1	4+1	gcc 4.8
Cortex A9	ARMv7	Odroid U3	4	4	gcc 4.8
Cortex A53	ARMv7	RPi3 Model B	4	4	gcc 6.3
ARM1176	ARMv6	RPi1 Model B	1	1	gcc 6.3

of the applications. Note that the default datasets in the provided C and CUDA version varies, hence the dataset sizes are modified to be equal, ensuring the application workload is consistent over all platforms. The precise platforms and used compilers are listed in table 6.1.

- **CPU:** In total 10 CPUs are included, 6 Intel CPUs and 4 ARM CPUs, to distinguish and compare the flexibility of embedded and desktop/server CPUs. The benchmarks are compiled with the GCC compiler [147]. Table 6.2 provides detailed information of the examined CPUs and the used compilers.

- **FPGA:** As PolyBench/ACC does not include applications described in hardware description languages, Vivado HLS [178] is utilized with it's default settings to transform C applications into register-transfer level (RTL) code, which can be directly targeted to Xilinx programmable devices. In Vivado HLS v2018.2 [178], when synthesizing a C function, a report is generated which provides performance metrics, such as resource utilization,

estimated clock period, loop latency, and function latency in clock cycles. To obtain more accurate estimates of resource utilization and the achieved clock period, the resulting designs have been synthesized towards the target FPGA platforms.

Unfortunately, when the application involves a variable loop bound, Vivado HLS fails to compute the iteration count required for performance analysis. An example of code where Vivado HLS v2018.2 fails is shown in code 6.1, where variable `k` causes the Vivado's loop analysis to fail for the loops L2 and L3.

```
1  for(k = 0; k < m; k++)                (L1)
2     for(i = k + 1; i < m; i++)         (L2)
3        for (j = k + 1; j < m; j++)     (L3)
4           A[i][j] = A[i][k] * A[k][j];
```

Code 6.1: Example loop-nest with bounds of L2 and L3 based on variable `k`

Fortunately this scenario occurs only in four out of 14 benchmarks, specifically *correlation*, *covariance*, *gramschmidt*, and *lu* as described in table 6.5. Nonetheless, for these four cases an alternative method is required to obtain performance estimates. A possible solution would be to use C/RTL Co-simulation in Vivado HLS, which simulates the application at the RTL level. However, for the selected benchmarks the simulation runtimes are prohibitively high, as well as the enormous amount of memory required for the simulation, which renders this option infeasible. Since the benchmark set targets polyhedral applications with strictly static control flow, manual derivation of the iteration counts is fortunately quite straightforward. Therefore, in this chapter manual static loop analysis is utilized to derive an *approximate* cycle count.

As an example, the latency of the loop-nest in code 6.1 can be manually derived as follows. The number of iterations of L2 and L3 depend on variable `k`, which only varies in L1. Thus, their iteration counts can be expressed as in equation (6.7), equation (6.8) respectively.

$$\#L2 = \sum_{i=1}^{m} (m - i) = \frac{1}{2} m(m-1) \tag{6.7}$$

Table 6.3: Overview of FPGAs.

Family	Device	LUTs	FFs	dsps	BRAMs*
Artix7	xc7a200t	129000	269200	740	730
Kintex7	xc7k480t	597200	597200	1920	1910
Virtex7	xc7v2000t	1221600	2443200	2160	2584
Zynq	xc7z100	277400	554800	2020	1510
Virtexuplus	xcvu13p	1728000	3456000	12288	5376
Kintexu	xcku115	663360	1326720	5520	4320
Zynquplus	xczu19eg	522720	1045440	1968	1968

* 17kB per BRAM.

$$\#L3 = \sum_{i=1}^{m} (m-i)^2 = \frac{1}{6} m(m-1)(2m-1) \tag{6.8}$$

By using an artificially small input, i.e., small `m`, these equations for all four affected benchmarks have been validated against short co-simulations and found to be exact for the unoptimized loop-nests. When vivado HLS optimizations are enabled through pragmas such as loop unrolling and pipelining, these equations have to be adjusted accordingly and again verified for small input sizes using co-simulation. These adjusted equations were all exact with the exception of the *gramschmidt* application, where the equations slightly deviated from the measured cycle count.

Aimed at exploring the impact of the amount and the type of resources on flexibility, Xilinx FPGAs from different device families, and with different resources, are included in this study. The Virtexuplus, for example, is an UltraScale+ version of the Virtex FPGA, with many more resources compared to a normal Virtex7. Table 6.3 lists the details of the selected FPGAs. All simulations were performed with Vivado HLS v2018.2 [178].

- **DSP:** DSPs are an important class of architectures that should be part of this study. Two multi-threaded Hexagon DSPs from Qualcomm, and one single-threaded DSP from Texas Instruments (TI) are therefore included. All measurements for these are based on simulations, as measuring on the actual devices was not available. The Hexagon V60 and V5 DSPs are simulated in the cycle-approximate mode provided by the Hexagon SDK [119], and Code Composer Studio v4 (CCSv4) [69] provides cycle-

Table 6.4: Overview of DSPs.

Processor	#Cores	L1I	L1D	L2	Simulator	Compiler
Hexagon V60	4	16K	32K	512K	Hexagon SDK	X
Hexagon V5	3	16K	32K	256K	Hexagon SDK	X
TI C6747	1	32K	32K	256K	CCSv4	X

accurate simulations for TI C6747. Table 6.4 provides more details of these three DSPs.

6.5.2 Benchmark Set

Selection of a benchmark set is orthogonal to the definition of flexibility given in this chapter. It depends on the application domain which benchmarks make sense. Since in this chapter a generic comparison between platforms is desired, a generic benchmark set is required. A restriction is the support of different platforms, which needs to be broad in this chapter for comparison between different architectures. Although many options exist, eventually PolyBench/ACC [44] was selected for its wide support of languages/platforms. Moreover it is a rather generic benchmark set, i.e., not specific to a specific application domain, and includes programs with static control flow which eases static analysis of the workloads. Furthermore, it provides multi-language versions of benchmarks, including C and CUDA, making it suited for cross-platform evaluation. In this chapter 14 applications from this set, supported by multiple languages, are evaluated using their standard dataset. It should be noted that the overall benchmark set contains more applications, but critical issues were encountered for several when targeting the selected platforms. In particular compilation issues led to blacklisting several applications. Table 6.5 provides details for each application.

6.5.3 Compiler Directives

It can be argued that not optimizing code gives a distorted image of reality, since programmers typically will spend some effort to manually optimize code for accelerators such as GPUs and FPGAs. As it is unfeasible to hand optimize all benchmarks for each platform, and the code quality would depend heavily on the programmer, a compromise is made by inserting compiler directives. Without directives, some compilers can hardly exploit the maximum potential of the

Table 6.5: Descriptions of the applied benchmarks from PolyBench/ACC.

Benchmark	Description
2mm	2 Matrix Multiplications (D=A.B; E=C.D)
3mm	3 Matrix Multiplications (E=A.B; F=C.D; G=E.F)
adi	Alternating Direction Implicit solver
correlation	Correlation Computation
covariance	Covariance Computation
doitgen	Multiresolution analysis kernel (MADNESS)
fdtd-2d	2-D Finite Different Time Domain Kernel
gemm	Matrix-multiply C=alpha.A.B+beta.C
gramschmidt	Gram-Schmidt decomposition
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
lu	LU decomposition
syr2k	Symmetric rank-2k operations
syrk	Symmetric rank-k operations

target systems. Therefore, to further investigate the impacts of applying compiler directives on flexibility, compiler directives are inserted in the C-code for CPU, FPGA, and DSP. The GPUs form an exception, as during the transformation to CUDA already manual effort has been made to optimize the code. Extra compiler directives would not change their performance in any significant way. The remainder of this section describes the various compiler directives used for each platform category.

- **CPU:** Multi-threading that parallelizes tasks among multiple threads is enabled by OpenMP directives. The outermost loop is parallelized in each kernel. In case dependencies between loop iterations prohibit parallelization, the (next) inner loop is parallelized.

- **FPGA:** For FPGAs directives are inserted to refine implementations, aiming at exploiting massive parallelism and increasing resource utilization. By default, Vivado HLS simply translates C functions into Verilog designs. Optimizations are applied barely without directives. For instance, Vivado HLS does not apply loop unrolling to the C code, meaning that one iteration of the loop is synthesized into a block of logic, which executes sequentially [178]. To enable some optimizations without modifying the code-base too extensively, `PIPELINE` directives were inserted to promote loop pipelining. When asked to pipeline a loop with an inner loop, Vivado HLS attempts to unroll all inner loops to enable the requested pipelining,

increasing resource utilization. If the inner loop spans many iterations, this may lead to a significant increase in required resources. Therefore pipelining of outer loops is used sparingly, only when the resulting design still fits all target devices, and performance does in fact improve. The default is to pipeline only the inner loop.

- **DSP:** To examine the hypothesis that best-effort techniques, including cache hierarchy and multithreading, negatively impact flexibility, Hexagon DSPs are simulated in two modes: timing accurate and inaccurate mode. With the accurate timing mode, Hexagon models cache, optimal multi-threading mode, and processor stalls. With the inaccurate mode, caches are assumed to be perfectly accessed, stalls are excluded, and a simplified multithreading model is simulated [120].

6.5.4 Intrinsic Workload Estimator

As discussed in section 6.4, the intrinsic workload of applications is used as the baseline to normalize against. In this section, a practical approach is proposed based on LLVM IR to extract the intrinsic workload from arbitrary applications. An embodiment of the described approach is made available as an open source tool [65] for anyone to use to estimate the intrinsic workload of their application.

In the compilation process that LLVM employs, applications written in diverse languages are translated by front-ends to a high-level intermediate language, IR. General optimization techniques are applied to this IR before generating target-specific code, allowing the reuse of optimization passes across different languages and targets. To serve this purpose the IR instruction set is relatively minimal, and more importantly, platform-independent. Thus, IR is selected as the elementary building blocks for approximating the minimal circuit of an application as detailed in section 6.4.

Figure 6.5 illustrates the procedure to automatically estimate the intrinsic workload of applications. Applications are first translated into LLVM IR by a front-end. Next, the instrumentation code is inserted by a custom LLVM pass called *libDynCountPass*, which triggers a callback for every execution of an IR instruction by the IR interpreter. These callbacks record dynamic instruction counts for each IR instruction type. As loops are fully unrolled in the ideal combinatorial circuit, computations related to control flow and memory accesses are excluded. This is achieved by skipping instrumenting these particular operations in LLVM IR. In particular, a *for* loop represented by IRs is composed

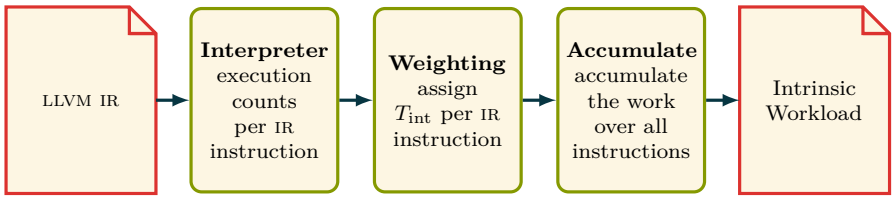


Figure 6.5: Automated intrinsic workload estimation using LLVM. Source code is made freely available [65].

of several basic blocks with dedicated names. Amongst blocks named *for.cond* and *for.inc* are meant for evaluating the loop condition and increasing the loop counter, respectively. When iterating through basic blocks in functions, the *libDynCountPass* pass skips instrumenting those blocks which names contain *for.cond* and *for.inc*. Furthermore, any parallelism in IR instructions due to vectorization is accounted for by passing the vector width to the instrumentation code. The instrumentation code then adjusts the dynamic instruction counts accordingly.

What remains is weighing all the IR instructions based on their equivalent circuit. To get the size of these circuits, the basic IR operations are mapped to basic 2-input-1-output and 1-input-1-output gates using Cadence Encounter RTL compiler [20]. The number of gates found for different IR instructions is listed in table 6.6. Note that the synthesis tool is set up to optimize for area, not speed. This choice results in a minimal number of gates to achieve certain functionality, rather than a speed-optimized design. In the case of an addition, for example, a carry look-ahead adder would be considerably faster than a ripple adder. For the minimal circuit though, the ripple adder is desired.

Instead of merely counting the number of basic gates per instruction, the argument can be made that in complementary metal oxide semiconductor (CMOS) technology some logic functions are harder to implement than others. An inverter only requires two transistors, where an XOR gate requires eight. To compensate for this the workload per IR is not only measured in gates, but also in the amount of transistors required to realize the circuitry. An approximation of the number of transistors per gate in CMOS technology is listed in table 6.7. We refer to the number of these transistors as the intrinsic transistors, or T_{int}, of an application. The intrinsic transistors per application is more precisely defined in section 6.5.4. Note that op$\in$Application in section 6.5.4 represents the *dynamically* executed

Table 6.6: Estimated hardware cost for spatial implementations of several IR instructions.

IR	32-bit			64-bit		
	Gates	Transistors	Depth	Gates	Transistors	Depth
add/sub[1]	188	880	63	380	1776	127
fadd/fsub	1905	8086	103	3494	14882	233
mul	5164	25458	130	20100	98476	313
fmul	4327	20490	97	16124	77246	215
udiv	4486	10763	1128	18217	44752	4350
sdiv	4777	21300	1189	18840	84616	4458
fdiv	12190	54146	992	66565	303284	2295
urem	4616	20426	1168	18452	82524	4440
srem	4882	21842	1228	19053	85682	4552
and	32	128	1	64	256	1
or	32	128	1	64	256	1
xor	32	256	1	64	512	1

[1] 16-bit *add/sub* operation: #(gate)=92, #(trans.)=432, #(depth)=31;
8-bit *add/sub* operation: #(gate)=44, #(trans.)=208, #(depth)=15;
1-bit *add/sub* operation: #(gate)=2, #(trans.)=12, #(depth)=1.

Table 6.7: Basic 2 input — 1 output logic gates and the number of transistors required to implement them in CMOS [122, Chapter 6].

Gate	AND	NAND	OR	NOR	XOR	XNOR	INV
Trans.	6	4	6	4	8	8	2

operations.

$$T_{\text{int}} = \sum_{\text{op} \in \text{Application}} \left[\sum_{\text{gate} \in \text{MinCiruit(op)}} \text{Transistors(gate)} \right] \tag{6.9}$$

An alternative to expressing circuit complexity in the number of gates or transistors is logic depth of the circuit. The rationale behind this is that the deeper the ideal circuit, the longer the minimum execution time would be. This approximation ignores parallelism inside the operation however, and would likely give rise to the selection of different circuits. For example, a straightforward ripple carry adder has the lowest gate count, but a relatively high logic depth. A carry look ahead circuit has a shallower logic depth, but increases the gate count. Which of these approximations to use is debatable. In this chapter the gate count is used,

as it bounds the designs to minimal compute effort for a particular operation. Nonetheless, logic depth could be an interesting alternative when computing the flexibility of high performance computing platforms for example.

6.5.5 Applied Methodologies

In practice benchmarking different systems in terms of performance, energy, and area efficiency is challenging, as it is difficult to unify results across different technology nodes. Furthermore, energy and area numbers are hard to obtain or measure accurately if the platform of interest is physically inaccessible, or not equipped for power measurements. Therefore, in several cases, we had to resort to extrapolation from publicly available data. For example, due to the lack of a power/energy measurement set up for each device, the thermal design power (TDP) of platforms is used to estimate their energy usage. The information used per platform and its sources are summarized in table 6.8, table 6.9.

The following list describes how each metric is obtained for the evaluated systems:

- **Normalised Performance:** Execution time, or cycle counts, can be reliably measured for each of the platforms. The measured times are normalised according to the estimated intrinsic workload of each benchmark, yielding normalised performance with the unit *number of intrinsic transistors per second*.

$$\text{Normalised Performance} = \left(\frac{T_{\text{int}}}{t_{\text{exec}}} \right) \tag{6.10}$$

  More intrinsic transistors that can be computed per time unit indicates higher performance. Note that some simulators only provide execution time in cycles, independent of frequency. This does not constitute a problem, since flexibility is invariant to frequency scaling.

- **Energy Efficiency:** Defined as the quotient of normalised performance and power, normalised energy efficiency is defined as *intrinsic transistors per joule*.

$$\text{Energy Efficiency} = \left(\frac{T_{\text{int}}}{t_{\text{exec}} \cdot \text{Power}} \right) \tag{6.11}$$

  The more intrinsic transistors that can be computed with a joule of energy, the higher energy efficiency.

  Energy numbers were estimated based on publicly available data. For most cases this means the reported TDP is used to estimate energy consumption.

For multi-core CPUs, this means the same TDP value is used for both single-threaded and multi-threaded mode. As the whole multicore processor is considered as a system, only utilizing a single thread in this system leaves other threads idle, resulting in energy and area overhead.

- **Area Efficiency:** Area efficiency is the quotient of normalised performance by area, expressed in *intrinsic transistors per physical transistor*. Here a 'physical transistor', or T_{phy}, refers to all the transistors implemented on the actual platform under test. This leads to the following definition of area efficiency:

$$\text{Area Efficiency} = \left(\frac{T_{\text{int}}}{t_{\text{exec}} \cdot T_{\text{phy}}} \right) \tag{6.12}$$

  By expressing area as the number of transistors on the device, and not the more typical μm^2, the area efficiency can be expressed independently of technology. Next to this it also allows an alternative interpretation of area efficiency, namely *(physical) transistor utilization*, which provides some insight into how many transistors are utilized effectively towards computing the application.

  Note that the transistor count of the FPGAs is for the whole device. When a design uses fewer resources, it could be argued that only the mapped transistors should be included. However, in this chapter, the full number of transistors is used, as those are physically always present, no matter if an application can utilize them, much like the transistors in any system.

- **Flexibility:** Compute system flexibility is derived relative to performance, energy, and area efficiency. However, due to the way energy and area are measured in this chapter, these three flexibility values yield the same ranking. In particular, for energy, the same TDP value is used for all applications on a given platform. Hence, the TDP is merely a scaling factor compared to performance, and flexibility is invariant to constant scaling of data points (in accordance with lemma 3). The same holds for the intrinsic transistor count in the area estimates, which again works out to be a constant scaling factor concerning performance. Hence in this particular case:

$$Flex_{\text{perf}} = GSD\left(\frac{T_{\text{int}}}{t_{\text{exec}}} \right) = GSD\left(\frac{T_{\text{int}}}{t_{\text{exec}} \cdot \text{Power}} \right) = GSD\left(\frac{T_{\text{int}}}{t_{\text{exec}} \cdot T_{\text{phy}}} \right) \tag{6.13}$$

Table 6.8: Specifications of processors in this chapter.

No.	Processor	Freq. (MHz)	Trans. (M#)	TDP (watt)	Node (nm)	Ref.
1	GTX TITAN	837	7100	250	28	[107][10]
2	GTX 570	732	3000	219	40	[107]
3	GTX 750 TI	1020	1870	60	28	[107]
4	Tegra K1	756	— [1]	14	28	[107][97]
5	Pentium 4	3400	169	84	90	[70]
6	i7-920	2670	731	130	45	[70]
7	i7-950	3070	731	130	45	[70]
8	i7-960	3200	731	130	45	[70]
9	i7-4770	3400	1400	84	22	[70][9]
10	i7-6700	3400	1750	64	14	[70][71]
11	ARM1176	700	—	2.9	40	[26]
12	Cortex A9	1700	—	4	32	[3]
13	Cortex A15	2300	—	5	23	[108][94]
14	Cortex A53	1200	—	4.4	40	[26]
15	TI C6747	300	22[2]	0.45[3]	65	[30][148]
16, 19[4]	Hexagon V5	650	—	—	28	[27]
17, 18[4]	Hexagon V60	2000	—	—	14	[121]

[1] "-" means no information available.
[2] Speculated based on TI C66x DSPs [30].
[3] Estimated by TI Power Estimation Spreadsheet 2013.3.
[4] Simulated in inaccurate timing mode as detailed in section 6.5.1.

Notably, this does not hold generically. When energy can be measured accurately across different applications and does vary, the equality no longer holds. Area efficiency flexibility strictly speaking would always be the same as performance flexibility, unless somehow only the active area of a system would be counted. Such a scenario may be meaningful in the context of multi-core systems executing a single thread. In this case, it may be preferable to exclude the inactive cores, resulting in a difference between area efficiency and performance flexibility.

6.5.6 Customized Processors

Apart from quantifying the flexibility of COTS systems, it is also interesting to investigate how flexibility relates to specialisation. To that end the work of Fisher et al. [33] is used, which reports speedups for a very long instruction word (VLIW) that is optimized to a varying degree for various target applications. More precisely, a baseline two issue slot VLIW processor with a 64 entry register file is

Table 6.9: Specifications of FPGAs used in this chapter.

Processor	Freq. (MHz)[1]		#Trans.[2]	Pwr (watt) [3]		Node
	no opt.	opt.	(M#)	no opt.	opt.	(nm)
Artix7	116	102	1025	1.4	2	28
Kintex7	120	103	2370	1.7	2.5	28
Virtex7	118	105	9700	2.3	3	28
Zynq	118	96	2200	1.7	2.6	28
Virtexuplus	118	104	14000	4.1	5.5	16
Kintexu	120	101	5300	2.1	3.4	16
Zynquplus	117	103	4100	2.1	2.8	16

[1] Average estimated clock period from Vivado HLS when the target clock period is 10 ns.
[2] Speculated based on a published value of Virtex UltraScale XCVU440[130].
[3] Estimated by Xilinx Power Estimator (XPE) 2018.2.2 based on the average design utilization of benchmark set.

constructed with one arithmetic logic unit (ALU) capable of integer multiplication, and one issue slot for accessing L1 or L2 memory. Starting from this baseline, an architecture exploration extending the architecture in several aspects such as ALU and multiplier count, memory ports, and register file size, is performed targeting each of the benchmark applications listed in table 6.10. This exploration is performed under a 'computing architecture cost' constraint, which is a custom metric defined in Section 3.3 of the work of Fisher et al. [33]. This cost metric depends amongst other parameters on the number of ALUs, width of datapaths, and number of ports to the register file. Within a given cost constraint, the VLIW architecture is first optimised targeting each individual application. For the ten benchmarks listed in table 6.10, this yields ten optimised architectures. Each of these architectures are still capable of executing the complete benchmark set, and as such a speedup can be reported for each application on each architecture. These speedup measurements are reported in the original paper [33, tables 8, 9, and 10]. Using these speedups, the flexibility of each resulting architecture can be computed, with the notable exception that normalisation is not based on intrinsic workload, but rather the single ALU baseline machine as described earlier.

Apart from just optimising for each application individually, Fisher et al. propose optimisation with a certain 'range'. When this range is set to $X\%$, it means the optimisation algorithm is allowed to sacrifice $X\%$ of the maximum performance

Table 6.10: Benchmarks for ASIPs taken from the work of Fisher et al.[33]. The benchmark applications in the lower part of the table are constructed out of the applications in the upper part. Benchmark naming is kept identical with the original work for consistency.

App.	Description
A	FIR symmetrical filter implemented using a 7×7 convolutional kernel.
C	Inverse DCT transform with dequantisation of the DCT coefficients. The algorithm used is the Arai, Agui, and Nakjima algorithm for scaled FDCT/IDCT, with some improvements, as described in [11, 164].
D,E	Colour conversion from the RGB to the YCbCr colour space (and vice versa, as described in the JPEG standard).
F	Halftoning via standard Floyd-Steinberg error diffusion (no stochastic weights update). The benchmark produces triplets containing 1 bit halftoned pixel.
G	1D bilinear scaling by integral factors along columns.
H	3×3 median filter using the standard algorithms not using a "smart" version of the median.
GF	1D bilinear scaling followed by Floyd-Steinberg halftoning.
GEF	1D bilinear scaling followed by E, a YUV $\leftarrow$RGB colour space conversion, followed by Floyd-Steinberg halftoning.
DH	RGB $\leftarrow$YUV colour space conversion followed by a 3×3 median filter.
DHEF	RGB $\leftarrow$YUV colour space conversion followed by a 3×3 median filter, followed by E, a YUV $\leftarrow$RGB colour space conversion, followed by Floyd-Steinberg halftoning.

of the application currently being optimized for, and trade that off for an improvement in the overall performance of the architecture over the complete benchmark set. For example, when the range is set to 0 %, the processor is allowed to have a cost of 10 according to the defined cost metric, and the optimization target is benchmark GEF, the maximum speedup for GEF is 8.93×. The harmonic mean performance of that tuned architecture over the entire benchmark set then equals 3.9×. However, when the range is set to 50 %, the optimisation procedure accepts a maximum penalty on the speedup of application GEF reducing it to 5.97×, in order to improve the harmonic mean performance over the complete benchmark set to 5.8×. These results can be verified in the original paper of Fisher et al. [33, table 9].

Using the reported speedups, the relation between this range parameter and flexibility can be investigated. The results and analysis of this experiment are presented in section 6.6.2.

6.6 Results and Analysis

The experiments are split into two categories. First flexibility measurements are performed for commercial of the shelf systems in section 6.6.1. Relations between flexibility and performance, energy efficiency, and area are investigated as part of this experiment. Furthermore the relations between flexibility and several architecture classes is examined. Secondly, section 6.6.2 analyses the relation between flexibility and processor specialisation, inspired by the work of Fisher et al. [33].

6.6.1 Commercial Off the Shelf Processors

This section presents the flexibility measurements of 25 platforms over 14 benchmarks, and investigates the hypothetical relations between flexibility, performance, energy efficiency, and area efficiency. In particular figure 6.6, 6.7, figure 6.8 respectively represent the relations between flexibility and these metrics. The indexing in the figures corresponds to the platform numbering in table 6.8. To achieve a fair comparison, performance results are scaled to the technology node of each platform. Accurate technology scaling is a topic on it's own, and different techniques should be used for different devices. For example, memories, wires, and gates all scale differently. To keep the comparison in this chapter straightforward, the basic assumption is used that performance, i.e., gate delay, scales linearly with the inverse of the technology node. Scaling under this assumption is sufficient to observe overall trends, and draw preliminary conclusions on the relation of the defined flexibility metric with respect to performance. Note that this technology scaling is an issue that arises in our particular measurement due to the lack of a set of platforms in the same technology node, and is orthogonal to the definition of flexibility itself. Any comparison of performance between architectures instantiated on different technology nodes requires such scaling.

Note that since performance, energy efficiency and area efficiency flexibilities are all equal for our measurements according to equation (6.13), the flexibility rankings do not move horizontally between figure 6.6, 6.7, and 6.8.

Before exploring the relations between flexibility and other metrics, it is interesting to note that the flexibility ranking has the power to discriminate between

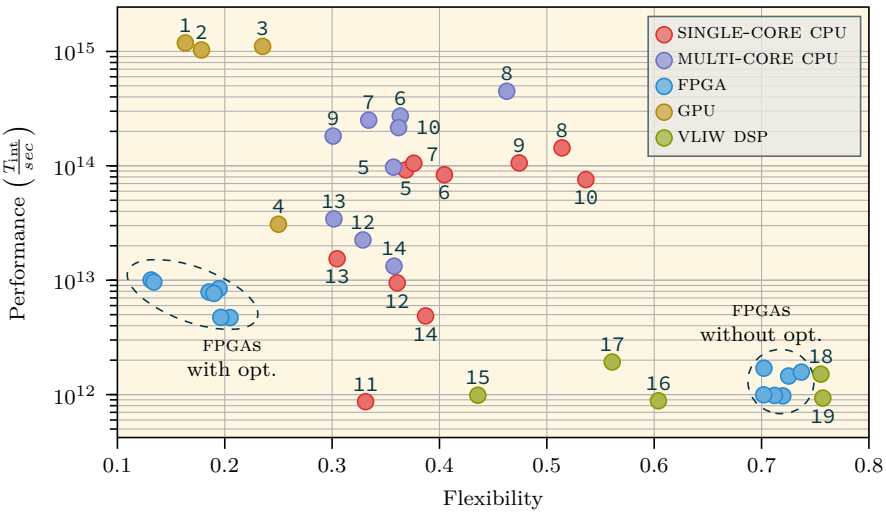


Figure 6.6: Performance and Flexibility. Platform indexes according to table 6.8. Some grouping of architecture classes in the flexibility dimension can be observed, suggesting various architectures generally indeed have a specific range of flexibility.

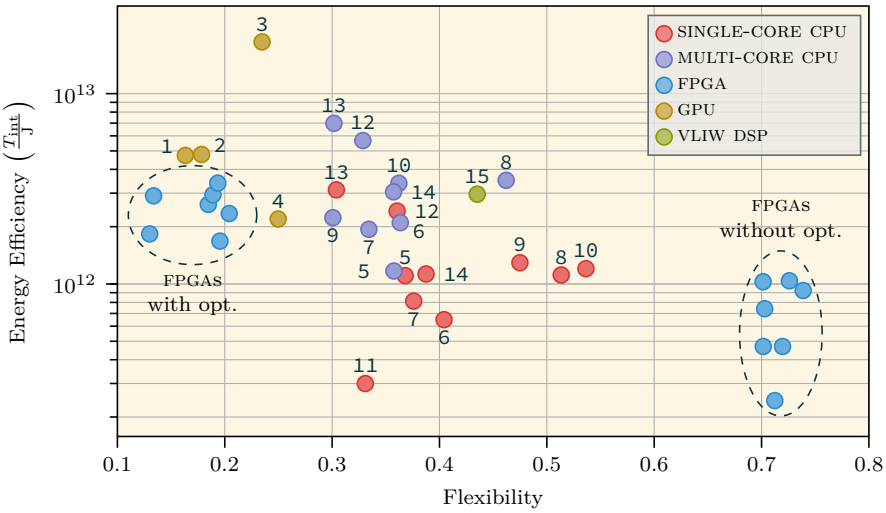


Figure 6.7: Energy efficiency and flexibility. Platform indexes according to table 6.8.

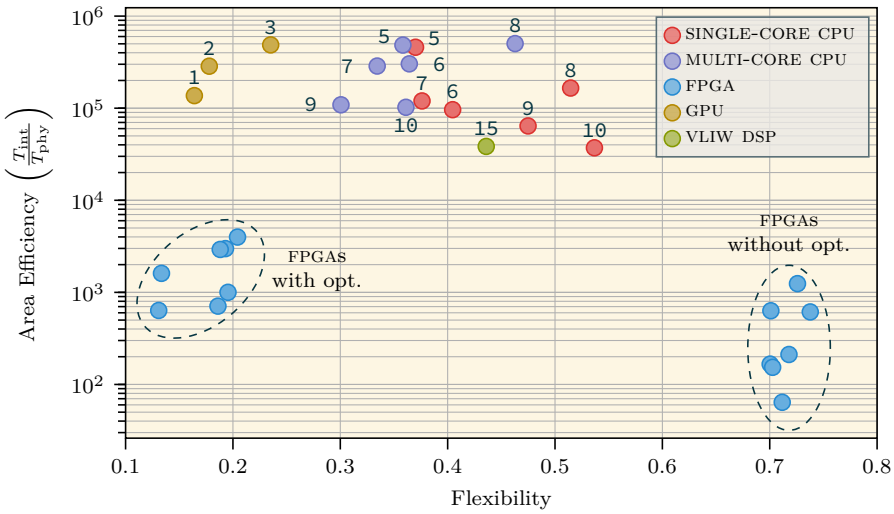


Figure 6.8: Area efficiency and flexibility. Platform indexes according to table 6.8.

different architecture classes. From our measurements we make the following seven observations:

1. The GPUs are clearly the least flexible of the tested architectures, while FPGAs without optimization are highly flexible. This does align with the idea that GPUs are specialized devices, supporting only a specific subset of algorithms (with DLP) very well, while FPGAs on the other hand are generic devices.

2. When optimization is turned on for FPGAs however, clearly some applications benefit more than others because of the extra resources, but without optimization their flexibility is far superior to other architectures.

3. Furthermore, multi-core CPUs measure as slightly less flexible than single-core CPUs, which again is intuitive as not all applications will benefit equally from extra cores. In general, it seems that for this generic benchmark set, architectures that employ more parallel execution pay a penalty in flexibility.

4. Predictably though, the same parallel architectures also have the highest performance, as can be seen in figure 6.6. This confirms the general notion

that there is a trade-off between performance and flexibility. Although there are definite outliers, overall higher flexibility implies lower performance. The ideal point in figure 6.6 is at the top right, combining high performance with high flexibility. The points closest to that corner are the general-purpose CPUs, showing these devices cover the middle-ground in this trade-off as would be expected.

5. The VLIW-DSP cores score very highly on flexibility with respect to other programmable architectures. This may however be an artifact originating from the measurement method. Where the other programmable devices have been evaluated on real hardware, this was not available for the tested VLIW devices. Instead simulators where used, which are claimed to be cycle accurate, but seemed to miss a memory model, and ignore other sources of variation that would be present in the real devices. Before drawing any firm conclusions regarding this class of architectures, these measurements should preferably be done on real hardware.

6. For energy efficiency (figure 6.7), a similar trend can be observed, although much less pronounced. In particular, unoptimized FPGA as the most flexible platforms have less of a gap to the CPUs in energy efficiency than they have in performance. A plausible explanation can be found in the much lower clock frequency of the flexible FPGA fabric, which obviously incurs a penalty in performance, but does not necessarily translate to low energy efficiency. Because the FPGAs essentially execute highly customized/parallel instructions, they may perform more useful work per cycle, leading to less register/memory overhead. The fact that CPUs execute their more generic, yet simple, operations much faster gives them a definitive edge in performance, however, the extra required cycles give them a relative handicap in energy efficiency. This shortage is overcame by the optimized FPGAs, which can customize their operations to achieve more work per cycle bringing them on par with the bulk of CPUs in terms of energy efficiency, even though their performance is lower.

7. In area efficiency (figure 6.8), the results are far less conclusive. The only outliers are the FPGAs, which are in a class of their own. This is to be expected, as the flexible FPGA fabric requires not only large silicon area for the lookup tables (LUTs), but also routing, which makes it very complex compared to other architectures. Between the other systems, the area efficiency numbers do seem to drop off slightly with increasing flexibility,

but the results are too close to draw any significant conclusions at this stage.

In general, the results show the proposed flexibility metric can distinguish between various architecture classes, indicating it represents a fundamental property. The measurements also align with some generally accepted notions surrounding flexibility, such as the trade-off between performance and flexibility.

6.6.2 Customized Processors

In this experiment the relation between flexibility and the measure of customisation/optimisation of an ASIP is investigated. In particular, the relations between flexibility, *computing architecture cost*, and *range* as defined in section 6.5.6 are of interest. This evaluation is based on the speedups reported by Fisher et al. [33, table 8, 9, and 10]. Using these speedups, the flexibility of each architecture is computed, the results of which can be found in table 6.11 table 6.12 in this chapter. Each row in these tables shows the speedups of all individual applications compared to the baseline VLIW processor, when the architecture is optimized for a particular application. The last two columns respectively give the harmonic mean speedup which Fisher et al. used as the cost function in their architecture search algorithm, and the flexibility as defined in this chapter. Note that when the 'range' parameter is set to infinite, it does not matter what the optimisation target application is, the optimisation algorithm will find only one architecture. After all, if it is allowed to compromise the speedup of the target optimization by an infinite amount, it does not matter what the target application is. The architecture with the best harmonic mean speedup over all applications will be selected regardless.

From the results in table 6.11 table 6.12, three key observations are made:

1. In general the high cost architectures achieve better generalisation than the low cost architectures. With plenty of compute resources available the benefits for each application average out, while with low compute cost only specific parts of all the applications benefit, leading to a more unbalanced speedup over the entire benchmark set.

2. For the architectures with cost ≤ 5, the flexibilities of the resulting architectures are almost all identical to the architecture found with range ∞. In fact, for most applications the selected architecture is actually equal, as can be seen in Table 9 of the original paper by Fisher et al. [33]. A clear outlier is application A, which when set as the optimization target actually

Arch.	**Speedup for application *X***										**HM**	**Flex.**
	A	C	D	F	G	H	GF	GEF	DH	DHEF		
Cost 5 — Range 0 %												
A	**6.12**	3.60	3.52	3.54	3.43	3.58	3.53	3.52	3.56	3.60	3.7	0.848
C	1.05	**3.93**	4.09	6.00	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
D	1.05	3.93	**4.09**	6.00	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
F	1.05	3.93	4.09	**6.00**	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
G	1.05	3.93	4.09	6.00	**5.72**	6.35	6.16	5.86	6.33	6.38	3.8	0.590
H	1.05	3.93	4.09	6.00	5.72	**6.35**	6.16	5.86	6.33	6.38	3.8	0.590
GF	1.05	3.93	4.09	6.00	5.72	6.35	**6.16**	5.86	6.33	6.38	3.8	0.590
GEF	1.04	3.93	4.09	4.53	5.72	6.15	6.14	**5.97**	6.31	6.36	3.8	0.593
DH	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	**6.33**	6.38	3.8	0.590
DHEF	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	6.33	**6.38**	3.8	0.590
Cost 5 — Range 10 %												
A	**6.12**	3.60	3.52	3.54	3.43	3.58	3.53	3.52	3.56	3.60	3.7	0.848
C	1.05	**3.93**	4.09	6.00	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
D	1.05	3.93	**4.09**	6.00	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
F	1.05	3.93	4.09	**6.00**	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
G	1.05	3.93	4.09	6.00	**5.72**	6.35	6.16	5.86	6.33	6.38	3.8	0.590
H	1.05	3.93	4.09	6.00	5.72	**6.35**	6.16	5.86	6.33	6.38	3.8	0.590
GF	1.05	3.93	4.09	6.00	5.72	6.35	**6.16**	5.86	6.33	6.38	3.8	0.590
GEF	1.05	3.93	4.09	6.00	5.72	6.35	6.16	**5.86**	6.33	6.38	3.8	0.590
DH	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	**6.33**	6.38	3.8	0.590
DHEF	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	6.33	**6.38**	3.8	0.590
Cost 5 — Range ∞ %												
All	**1.05**	**3.93**	**4.09**	**6.00**	**5.72**	**6.35**	**6.16**	**5.86**	**6.33**	**6.38**	3.8	0.590

Table 6.11: Speedup and flexibility of ASIP architectures with cost ≤ 5, based on Fisher et al. [33]. **HM** denotes the mean harmonic speedup.

Arch.	**Speedup for application *X***										**HM**	**Flex.**
	A	C	D	F	G	H	GF	GEF	DH	DHEF		
Cost 15 — Range 0 %												
A	**13.06**	5.88	3.52	5.63	4.95	9.68	8.13	8.65	9.60	9.14	6.8	0.690
C	11.04	**7.46**	3.86	5.25	5.41	10.50	8.39	8.93	10.61	9.88	7.2	0.710
D	10.72	6.07	**4.42**	6.13	5.42	6.35	6.16	5.86	6.31	6.38	6.1	0.811
F	10.72	6.07	4.42	**6.13**	5.42	6.35	6.16	5.86	6.31	6.38	6.1	0.811
G	9.38	6.15	4.33	6.13	**5.72**	6.35	6.16	5.86	6.33	6.38	6.1	0.838
H	5.95	7.46	3.86	3.98	5.41	**10.52**	5.75	6.79	10.58	9.74	6.2	0.705
GF	11.04	7.46	3.86	5.25	5.41	10.50	**8.39**	8.93	10.61	9.88	7.2	0.710
GEF	11.04	7.46	3.86	5.25	5.41	10.50	8.39	**8.93**	10.61	9.88	7.2	0.710
DH	11.04	7.46	3.86	5.25	5.41	10.50	8.39	8.93	**10.61**	9.88	7.2	0.710
DHEF	10.54	6.43	3.86	5.25	5.41	10.50	8.39	8.93	10.55	**10.06**	7.1	0.710
Cost 15 — Range 10 %												
A	**13.06**	5.88	3.52	5.63	4.95	9.68	8.13	8.65	9.60	9.14	6.8	0.690
C	11.04	**7.46**	3.86	5.25	5.41	10.50	8.39	8.93	10.61	9.88	7.2	0.710
D	10.72	6.07	**4.42**	6.13	5.42	6.35	6.16	5.86	6.31	6.38	6.1	0.811
F	13.06	5.88	3.52	**5.63**	4.95	9.68	8.13	8.65	9.60	9.14	6.8	0.690
G	11.04	7.46	3.86	5.25	**5.41**	10.50	8.39	8.93	10.61	9.88	7.2	0.710
H	11.04	7.46	3.86	5.25	5.41	**10.50**	8.39	8.93	10.61	9.88	7.2	0.710
GF	11.04	7.46	3.86	5.25	5.41	10.50	**8.39**	8.93	10.61	9.88	7.2	0.710
GEF	11.04	7.46	3.86	5.25	5.41	10.50	8.39	**8.93**	10.61	9.88	7.2	0.710
DH	11.04	7.46	3.86	5.25	5.41	10.50	8.39	8.93	**10.61**	9.88	7.2	0.710
DHEF	11.04	7.46	3.86	5.25	5.41	10.50	8.39	8.93	10.61	**9.88**	7.2	0.710
Cost 15 — Range ∞%												
All	**11.04**	**7.46**	**3.86**	**5.25**	**5.41**	**10.50**	**8.39**	**8.93**	**10.61**	**9.88**	7.2	0.710

Table 6.12: Speedup and flexibility of ASIP architectures with cost ≤ 15, based on Fisher et al. [33]. **HM** denotes the mean harmonic speedup.

yields a very high flexibility. As can be seen in the table, the speedup of A is rather significant, while the other benchmarks appear to benefit fairly equally. The conclusion must be that application A is quite different from the other benchmarks in the set, and optimising for it within low cost constraint does not benefit the other applications. In the original paper of Fisher et al. [33] it can be seen that the number of registers for architecture A is higher than the other architectures, suggesting that application A benefits heavily from more registers, while that does not help the other benchmarks that much.

3. A higher range interestingly does not always result in a more flexible architecture, as is the case for application G at cost ≤ 15 in table 6.12 for example. The sacrificed performance of G when increasing the range from 0 % to 10 % does lead to a higher harmonic mean performance, but the variation in speedups increases. This again demonstrates that performance and flexibility are orthogonal properties, and while Fisher et al. optimised for overall performance, the flexibility decreased. If the goal of an architect is to design a processor that is likely to perform well under varying applications, the optimisation goal should thus have been flexibility and not overall performance.

Note that item 1 is supported by the findings of Arnold and Corporaal [12], who investigate the benefit of adding custom instructions to a processor that replace two basic operations. Figure 6.9 is taken from their work, and shows the reduced operation count by adding a library of size x with more complex operation patterns of size two, i.e., replacing two operations in the original execution graph. The theoretical best case is an operation count reduction of 50 %. As can be seen in the figure, when the number of added complex patterns is on the low end, e.g. 10 patterns, the vertical spread is relatively high. A high vertical spread equals low flexibility, since there is large variation between different applications. To validate this the operation reductions for 10 and 40 patterns were extracted from the image[2], and summarized in table 6.13. It shows that overall flexibility increases from 0.83 to 0.95 when adding 30 extra patterns. This corresponds to the cost ≤ 5 architectures from Fisher et al. with relatively limited resources, and low flexibility. When the number of available resources increases however, such as the cost ≤ 15 architectures or 30+ patterns, the flexibility increases.

[2]The original work [12] does not list the raw numbers, but a vector image of the graph could be recovered from the pdf file which allowed accurate reconstruction of the measurements.

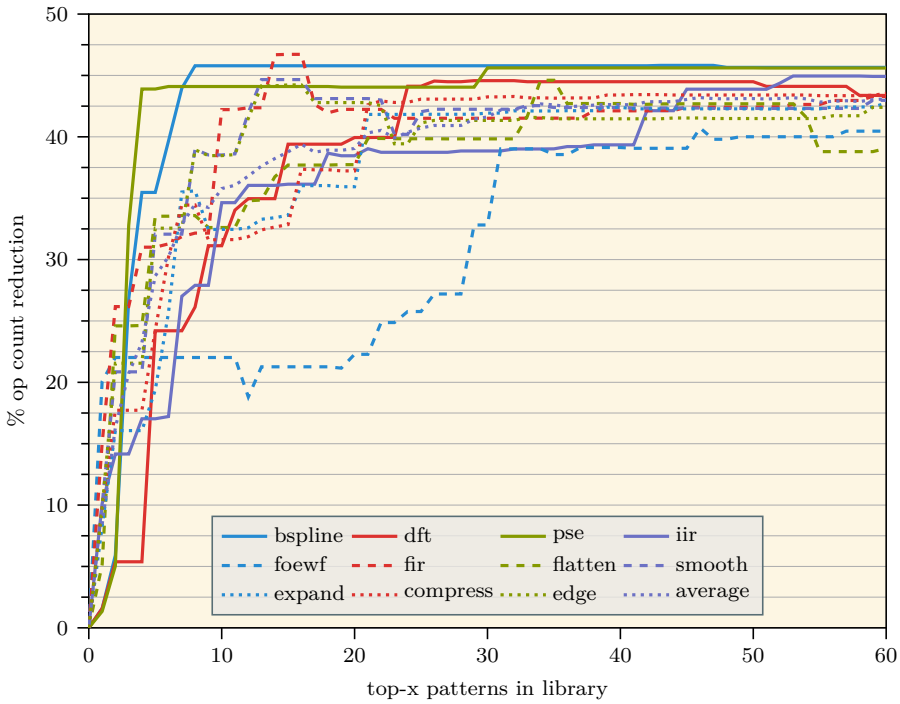


Figure 6.9: Operation count reduction by using a library with the x most used patterns with x ranging from 0–60 [12]. After 10-15 patterns, most applications start to asymptotically approach their maximum.

With more resources available, there is more room to have every application profit maximally.

6.7 Comparison with Existing Definitions

This section compares the proposed flexibility metric with related work that provide alternative definitions. In particular the proposed method is compared against versatility as defined by Van Berkel [77], and 'VersaBench versatility' as defined by Rabbah et al. [123], in section 6.7.1 section 6.7.2 respectively.

Table 6.13: Operation count reductions from figure 6.9, and associated flexibility.

	Top X	
App.	10	40
bspline	45.8	45.8
dft	31.1	44.5
pse	44.1	45.6
iir	34.6	39.3
foewf	22.0	39.1
fir	42.2	42.1
flatten	32.6	42.7
smooth	38.5	42.4
expand	32.5	42.4
compress	31.6	43.4
edge	38.5	41.5
Flexibility	0.82	0.95

6.7.1 Flexibility and Versatility

In the search for alternative definitions of flexibility in section 6.2, the qualitative and quantitative definitions of *versatility* provided by Van Berkel [77] appeared to be most related. Rather than looking at change of a system's performance metrics under the influence of external changes, the assumption is made that the less information required to specify an amount of work to a system, the less versatile it must be. In terms of computing systems this translates to the number of dynamic (instruction) bits required to execute a given program on a given processor. The fewer bits required, the fewer options were available to select functionality from, hence the less versatile the system must be. This is captured in the versatility formula in equation (6.1).

This interesting approach does not directly measure the effects of external changes on the system. Rather, it makes the underlying assumption that more functionality to select from *should* result in a more flexible system. After all, if there are more options/instructions, it is easier to adapt to a new program. The danger in this assumption is that the computer architect now has become part of the metric. I.e., it is assumed that the extra added functionality is diverse enough to handle more cases. In a way, the number of dynamic bits per workload is a measure of both how flexible the system is, and how well the architect anticipated and addressed possible changes to the system.

Therefore we still reason the qualitative definition provided at the start of this section is more suited for defining a flexibility metric, and versatility is in fact different yet very related. In particular, the difference between measured versatility and flexibility is an indication of how well a system architect has designed the system. When a system is not flexible, but highly versatile, apparently a price is paid for having more options/functionality, but it did not translate into added flexibility. In fact, the ratio between flexibility and versatility can be regarded as a measure of success of an architect to balance the cost of instruction size with return in flexibility.

Given this conclusion it is interesting to compare flexibility and versatility for various architectures. Application of the versatility metric is slightly more involved than the proposed flexibility metric however for two reasons:

1. Versatility is only defined per application, and as such for different applications one architecture would have multiple flexibilities. To be able to compare we therefore propose to use the geometric mean of all versatilities of a benchmark set to obtain a single flexibility number per architecture.

2. The definition of versatility includes the number of *useful operations*. This term is not defined exactly by Van Berkel. In this analysis we will therefore use the proposed intrinsic workload instead, which also should yield a fair comparison between versatility and flexibility.

The resulting definition of Versatility as used in this comparison is given in equation (6.14), where x_i is an application in benchmark vector $\boldsymbol{X}$, and $W_{int}(x_i)$ is its intrinsic workload.

$$Versatility(\boldsymbol{X}) = \left(\prod_{i=1}^{n} \frac{avg\_instruction\_size(x_i)}{W_{int}(x_i)/\#instructions(x_i)} \right)^{\frac{1}{n}} \tag{6.14}$$

Finding the number of instructions on Intel and ARM machines is straightforward by merit of the available performance counters. The average instruction size for the considered ARM devices is also simple, a fixed 32 bits. The instruction size of the Intel machines is dynamic on the other hand, and not monitored by hardware performance counters. Therefore an estimate of 20 bits per instruction on average is used for the Intel machines as per the work of Ibrahim et al. [67].

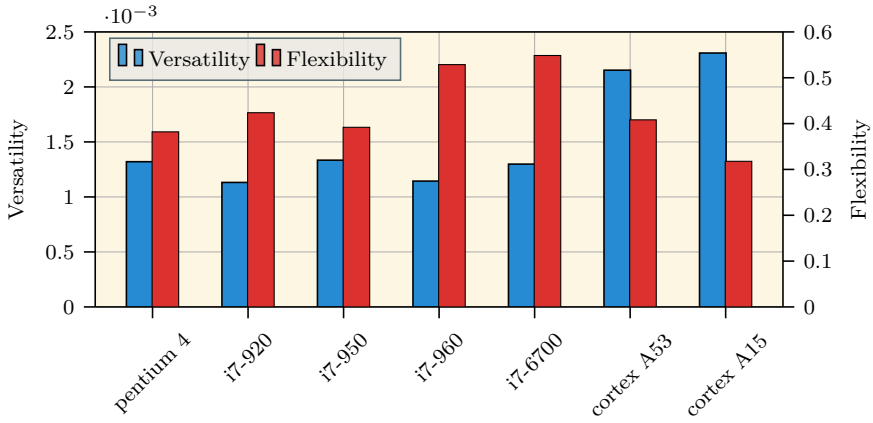


Figure 6.10: Flexibility and versatility comparison between Intel and ARM CPUs. Notice the double y-axis. Because of their very different ranges, flexibility and versatility can not be compared one-to-one. Instead, focus on the relative differences between applications for both metrics.

Figure 6.10 shows a side-by-side comparison of performance flexibility with versatility for several Intel and ARM CPUs. The results are quite interesting, and two observations can be made:

1. First the flexibility of the Intel machines increases with newer generations, while the versatility is more or less equal. This implies that the designs improve in such a way that without spending more instruction bits, the Intel processors have become more flexible.

2. Second, the ARM processors have a much higher versatility than the Intel processors, yet fail to capitalize on this in particular compared to the later generations of Intel processors.

This outcome seems to make a case for variable instruction width as opposed to fixed.

However, not covered here is the use of the 16-bit instruction set of ARM, which may paint a different picture. Such investigations are left as future work, but it can clearly be concluded that by having quantitative metrics insight can be gained, and processor design can potentially be guided by such metrics to truly

get to machines that balance average instruction size, performance, energy, and flexibility.

6.7.2 Flexibility and VersaBench Versatility

Apart from versatility as defined by Van Berkel [77], there is a variant defined by Rabbah et al. [123] as discussed briefly in section 6.2.2 As the motivation behind this second definition is also close to the goals of this chapter, this section provides a short comparison between flexibility as defined in this chapter, and "VersaBench Versatility".

To understand the differences between VersaBench Versatility and flexibility, it makes sense to look at the properties of flexibility as defined in section 6.3.2. In particular, we state that flexibility should be orthogonal to performance. The motivation behind this property is illustrated in figure 6.2, where the most performant system shows higher variation to change than the slowest system.

VersaBench Versatility on the other hand normalizes performance based on the fastest processor known for each application, and uses this to rank processors. Therefore, performance and VersaBench versatility are directly related. In fact, it can be shown that the normalisation by the fastest processor for each application is irrelevant for the final versatility **ranking**. Lemma 4 lemma 6 show that the normalisation baseline is cancelled out when calculating the ratio of two positive datasets as shown in equation (6.15), where $\boldsymbol{X}$, $\boldsymbol{Y}$ and $\boldsymbol{B}$ are positive datasets.

$$\frac{GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right)}{GM\left(\frac{\boldsymbol{Y}}{\boldsymbol{B}}\right)} = \frac{GM(\boldsymbol{X})}{GM(\boldsymbol{Y})} \tag{6.15}$$

$$\frac{GSD\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right)}{GSD\left(\frac{\boldsymbol{Y}}{\boldsymbol{B}}\right)} \not\equiv \frac{GSD(\boldsymbol{X})}{GSD(\boldsymbol{Y})} \tag{6.16}$$

Hence, the ranking obtained using the metric proposed by Rabbah et al. is equivalent to ranking on average performance. For completeness, flexibility as defined in this chapter does depend on the chosen baseline, as shown in equation (6.16), supported by lemma 8: $GSD\left({}^{\boldsymbol{X}}/_{\boldsymbol{B}}\right) \geq {}^{GSD(\boldsymbol{X})}/_{GSD(\boldsymbol{B})}$. Note that '$\not\equiv$' here denotes that the relation does not necessarily hold. In conclusion, VersaBench versatility is in fact a ranking based on average performance, while flexibility is truly an orthogonal property.

6.8 Discussion & Open Issues

The work presented in this chapter is an attempt at defining a flexibility metric for processors. The lack of ground truth, however, combined with several existing related studies presented in section 6.2, entails that the resulting definition is to be placed into context for it to carry any meaning. The definition presented in this chapter is not to be taken as final, as many open questions underlie its definition to which there are no definitive answers yet. This section discusses these open issues, and how they were handled in this chapter.

6.8.1 From qualitative to quantitative

To systematically identify open issues and the choices made when deriving a quantitatively metric form the qualitative definition of flexibility, we identify three key components in the qualitative definition of flexibility as given in section 6.3.1:

1. The measured system and its set of (changing) external inputs (S_i)
2. Observed Performance Metric (m)
3. Measure of Affectedness ($f(S_i, m)$)

Each of these components is to be mapped to computing systems to properly define computing system flexibility. However, each component leaves room for different interpretations, which is the root cause of different definitions of processor flexibility in related work. What follows is an attempt to capture different interpretations of these terms, and motivate the choices made in this chapter.

1. **System and External Inputs**
   Defining what the measured system and its (changing) inputs, S_i, seems trivial, but turns out to be both complex and very relevant for the resulting definition. For example, if the system is defined as a bare processing core, then the system's external inputs would be machine instructions and data. Changes in the instruction and data stream influence the energy usage of the core in different ways, and flexibility may measure the sensitivity of this energy usage for different instruction streams on a per cycle basis. However, if the system includes not only the core but also an instruction cache, the fine grain energy consumption may already vary without new instructions coming in from *outside* the system. In this case, some forms

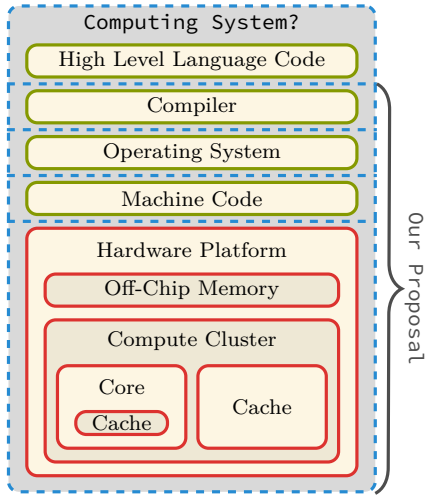


Figure 6.11: Where to draw the line, what is part of a compute system, and what is not?

of temporal integration has to be used, and flexibility measures the effects of new blocks of instructions loaded on this averaged energy usage. The key message is that the cache can be part of the system, and the cache size can also influence the system's flexibility in this case.

In fact, many components in the chain from application idea in the mind of a developer, up to final execution on a core can influence the system's performance metrics. Therefore, each of these components *could* be seen as part of the computing system, as illustrated in figure 6.11. Although not shown in the figure, in an extreme case the programmer who writes a program according to changing specifications could even be considered part of the system. The supported language, compiler, memory hierarchy, and processor architecture then all influence the measured system. Note that the world is much larger than what is captured in figure 6.11. For example, loop-buffers, or configuration memory in an FPGA could all be different points to draw a system boundary.

Where the system boundaries are defined is rather arbitrary from this viewpoint, although it would be sensible to not make an individual programmer part of the system. Yet, it is preferable to stay at a higher level, as the lower levels quickly become more specific for a certain subclass of

systems, e.g., configuration memory for FPGAs. Therefore we have chosen to stop at the compiler level, where the compiler is still considered part of the system, and the source code is the external input. In particular, the source code of different applications, which is also the external input used for most commonly used processor benchmark suites. In these benchmarks, the applications are taken as the changing input, hence this best practice is followed in this chapter. This choice also aligns with the view of Zse et al. [145], who consider the mapper (compiler) part of the system.

2. **Observed Performance Metrics**
   There exist many widely used metrics in computer system design, such as energy-efficiency, area-efficiency, and runtime. In principle, any of these or even a combination, can be selected as the observed performance metric. Related work typically ties flexibility exclusively to runtime, e.g., the work of Rabbah et al. [123]. We argue this is too restrictive, and a plurality of meaningful flexibilities can be defined. Zse et al. [145] already hint at flexibility in terms of performance and energy efficiency, which is more in line with the reasoning of our work. In particular, it is worth noting that flexibility in our view as such is a derived metric, since it measures changes in other primary metrics. For any primary metric, flexibility can be defined.

3. **Measure of Affectedness**
   Whereas for the other two terms, there is a history of common practices in computer system design to build upon, it is the "measure of affectedness" ($f(S_i, m)$) about which there is the least consensus in the community. There are numerous ways to quantify changes in a metric m. If the flexibility function f is to be generic for any metric m however, it is clear the changes in m caused by changing inputs have to be normalised. The selection of a normalization method is another degree of freedom, which is part of defining f.

   In the work of Rabbah et al. [123], the runtime is normalised by comparing the runtime of applications to the best-known runtime over all processors. Change in the runtime of a processor compared to this "optimal runtime" is seen as inflexible. The total change over a selected benchmark set is then seen as the flexibility of a processor. Thus, higher absolute performance over the benchmark set is taken to mean higher flexibility. Although this direct coupling of performance and flexibility may seem appealing, in particular for a designer who needs to design or select a system with

flexibility as a metric, we argue that such reasoning is a fallacy. The implication that the most performant machine automatically is the most flexible machine is unfounded, and the selection as the best-known runtime for each application is an arbitrary baseline for a flexible machine.

Instead of normalizing against the best known runtime, we postulate there must be a notion of intrinsic workload for each application that can be normalised against. The definition of this intrinsic workload again poses several challenges, which is further discussed in section 6.8.2.

Finally, we argue that *any* change in intrinsic workload normalised metric m, either positive or negative, makes a system less flexible. As a measure of how affected a metric m is under a set of changes, *variance* seems a natural choice to us.

6.8.2 Intrinsic Workload

In section 6.4, the term *intrinsic workload* is introduced, which refers to the notation that an application inherently describes a fixed amount of work. An open question is, assuming the notion of such a fixed amount of work per application is correct, how to properly define this intrinsic workload. There are many possibilities, and selecting one that is both theoretically and practically appealing is a difficult task. A fundamental approach may consider Landauer's principle [87], which states there is a minimum amount of energy that is dissipated when a bit of information is erased. For a typical irreversible computation that consumes two operands and produces one output value, this principle can be used to compute a minimum amount of work for that operation. However, if the computation is reversible, no information is lost in the system, and theoretically no energy would be required to perform such a computation. Thus, if an application is expressed in reversible operations, it may not have an intrinsic workload at all, and computation may in fact be free. This is the promise of the field of reversible computing, and maybe the only fundamentally correct answer to the question of how much workload any given application represents, zero.

This definition of (the non-existence of) intrinsic workload from a physics perspective does not provide any insight for the practical machines in current technology however, so for practical reasons, a more pragmatic approach is taken in this chapter. As extensively described in section 6.4, an approach is chosen which expresses workload in terms of the size of the minimal circuit that implements an application. The motivation for this approach is that it automatically weights

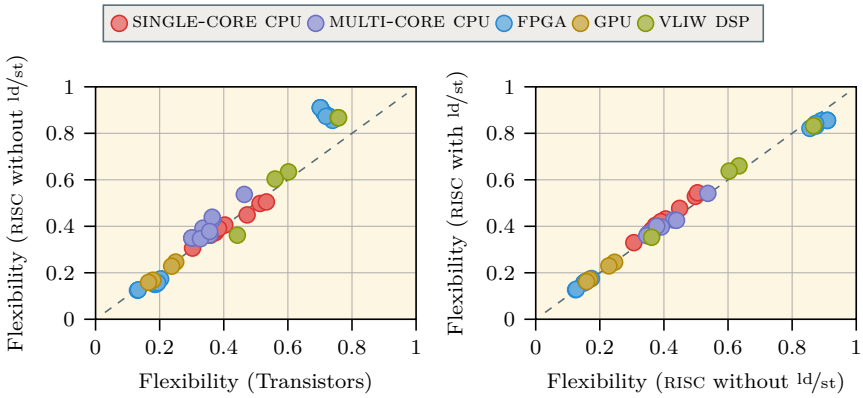
operations based on their complexity, and a multiplier circuit will require more transistors than an adder.

A downside of this choice is the infeasibility to construct the schematic of a truly minimal circuit for any application. Such a circuit would be extremely large, and logic minimization is proven to be NP-complete [19]. As discussed in section 6.5.4, the practical choice was made to approximate the size of the ideal minimized circuit by dividing applications into LLVM IR instructions and weighting those based on their approximated minimal circuits. This choice is very much motivated by the desire to develop a flexibility metric that is also applicable in the real world and not just a theoretical notion. In particular, this approximation may be done in various different ways, and remains an open topic of research.

RISC versus Transistors

In section 6.4 the choice is made to express workload in intrinsic transistors. It can be questioned though how much this refinement of RISC-like operations to intrinsic transistors impacts the resulting flexibility measure. To investigate this, figure 6.12a plots flexibility based on intrinsic transistors (horizontally) versus flexibility based on RISC-like operations. The flexibility based on RISC-like operations is calculated similarly to the proposed transistor-based flexibility, except that all operations in table 6.6 are set to one. If both flexibility metrics are exactly the same, the points in figure 6.12a would be on the diagonal of the plot. Diversion of points from the diagonal indicates differences between the two metrics.

As can be seen in the figure, the majority of the points are close to the diagonal, indicating the refinement towards transistors does not change the flexibility significantly. Only the most flexible points, which represent the unoptimized FPGAs, seem to be classified as significantly more flexible when normalised to RISC instructions rather than transistors. A possible explanation for this phenomenon is that the FPGA uses its DSP slices to perform the multiplications, relatively lowering their performance complexity compared to other operations. As such, weighting the multiplications as more work based on the circuit complexity may expose some inflexibility of the unoptimized FPGA solutions. For the RISC baseline this skewing is not present, and hence the solutions are quantified as more flexible. On the optimized FPGA designs, higher degrees of parallelism may hide this effect. It is however difficult to reason about such effects. Nonetheless it can be concluded that although the refinement into transistors is from the



(a) Transistor based vs RISC operation based flexibility. **(b)** Impact of accounting for load/stores.

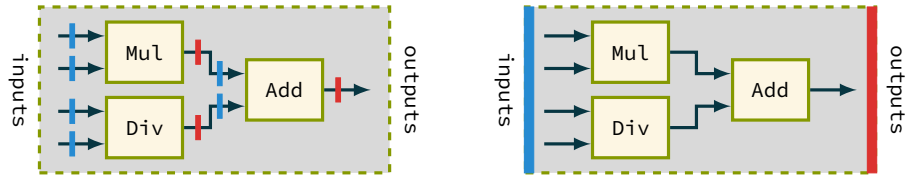
Figure 6.12: Effect of different normalisation strategies.

theoretical viewpoint arguably "correcter" then not accounting for operation complexity, omission of this refinement for practical considerations would not have a large impact on the measured flexibility.

Loads and Stores

Although the use of an approximate minimal circuit makes the definition of intrinsic workload practical for real machines, one problem mentioned at the start of this section still remains. The minimal circuit of a matrix transpose algorithm would not involve any transistors, and would consist only of wires, i.e., loads and stores do not exist. Although this may be a fair game from the theoretical point of view, the retrieval or storage of information should not need to cost any energy, for practical purposes it may not be the most workable approximation. An alternative that is possible within the proposed intrinsic workload estimation framework is to weight IR based loads and stores.

How these loads and stores are to be weighted is again a point of discussion. One additional practical consideration could be to weight external and internal loads differently, since external memory accesses are typically much more expensive than internal memory accesses. A very crude way of separating the two would be to count an internal load for each input operand of an operation, and a store for each output operand. External loads are then defined by the input



(a) Arithmetic operations + internal memory operations (2 loads and 1 store for each operation).

(b) Arithmetic operations + external memory loads and stores.

Figure 6.13: Two conceptual combinatorial-circuit models with loads (grey) and stores (blue).

to an application that has to be loaded once, and the output produced by the application which has to be stored once externally. These options to weighting loads and stores are illustrated in figure 6.13. Other approaches, like taking reuse distance into account to decide on internal versus external memory accesses, are again possible, although with each more practical consideration for memory levels and technology guided design it becomes more polluted with memory architecture specifics.

Figure 6.12b plots the flexibility based on RISC operations with, and without loads and stores. For this evaluation, the RISC based metric is used, as it allows loads and stores to be simply weighted by one. This avoids the difficult problem of weighting the loads and stores in terms of transistors, which could, depending on the approach chosen, yield a very different flexibility ranking. Instead, figure 6.12b shows that, when loads and stores are simply counted as a single RISC instruction, their impact on the resulting flexibility metric is minimal for the selected benchmark set. It should be noted that this observation is dependent on the evaluated benchmarks. For a matrix transpose algorithm, for example, the outcome is expected to be completely different.

6.8.3 The most flexible machine?

When defining flexibility, a natural question to ask is "what is the most flexible machine?". After all, flexibility sounds like a desirable property, so *the most* flexible machine must be quite an impressive device. Based on the definition of flexibility provided in this chapter, we argue the opposite however, and claim that the most flexible machine is mainly of academic interest, rather than a

viable design target. For practical machines, instead, a balance should be struck between flexibility and other system properties.

Before diving into this question, it should be noted that when following the strict definition, there is no such thing as *the* most flexible machine. Since flexibility is scale-invariant as shown in section 6.3.2, *the* most flexible machine can always be slowed down by a real values factor S, leading to not one, but an infinite series of 'most flexible machines'. For the intents of this section however, we will consider that the most flexible machine from the question is not slowed down, i.e., the question really is "what is the most performant of all the most flexible machines?".

When deriving this most flexible machine, the first step is to consider the normalisation step of the flexibility definition. It is evident that when a machine's runtime matches the intrinsic workload, apart from some scaling factor, the metric will yield a flexibility of exactly one. This clearly exposes the dependency between the choice of definition of the normalisation workload, and the definition of the most flexible machine.

When the practical intrinsic workload definition based on weighted LLVM IR instructions is applied, the most flexible machine is exactly the machine that implements those instructions, and varies its execution time depending on the selected weight. This execution time may be scaled with some factor, as long as all operations are scaled with this same factor. The most flexible, most performant machine would therefore take the slowest operation of the LLVM IR instructions it implements, and slow down all other operations to match the selected weighing. This would automatically lead to a machine with a flexibility of one.

The selection of LLVM IR as a baseline was done for practical purposes however, and is rather arbitrary in this theoretical discussion about the most flexible machine. Instead, when the definition is brought back to its essence of the gate-count of the smallest possible circuits of elementary gates that implements the application, the most flexible machine changes drastically. The separation into high-level mathematical and logic operations disappears, and the choice for operand sizes of powers of two disappears. To be 'the most flexible' machine under this definition, the simplest approach is perhaps to build a machine which as operations implements all elementary two-input,one-output logic gates. By computing one gate at a time, this can exactly mimic the dedicated circuit, although serialized over many clock cycles. Any attempts to speed this machine up, by exploiting intra-word parallelism for example, would result in logic that

speeds up some operations, but not all, reducing the flexibility. According to Amdahl's law, programs can be divided in a sequential and parallel part. Unless this sequential part does not exist at all for the considered benchmark set, any attempts to exploit parallelism favour only a subset of the program, reducing flexibility. The most flexible machine is therefore a very rudimentary, and in terms of performance arguably boring device, which performs only a single elementary gate operation at a time.

6.9 Conclusions

The term flexibility is frequently used in computer architecture literature [75, 76, 105, 171, 126], despite the lack of both a proper qualitative and quantitative definition. This is a harmful situation which leads to contradictory statements regarding flexibility as a property and its relation to other system metrics, and as such does not advance knowledge of computer architectures, but rather dilutes fundamental reasoning. In an attempt to address this problem, a survey of compute system flexibility as used and defined in literature was performed in order to collect general ideas about flexibility in the community, and hypothesized relations between other metrics such as performance and energy efficiency. Furthermore existing definitions of flexibility and related notions were collected and classified. Based on these statements regarding flexibility in literature, first a qualitative, and consequently a quantitative definition of flexibility in computing systems was derived. As part of the quantitative definition, intrinsic workload is introduced as a generic method of normalizing applications. An accompanying open source tool was released to automate the estimation of intrinsic workload [65]. Using this tool, flexibility is evaluated on 25 platforms over 14 benchmarks, validating that the proposed metric conforms with some commonly accepted notions of flexibility.

Globally, the proposed flexibility metric orders some major architecture classes from least flexible to most flexible as: GPU, CPU, DSP[3], and FPGA. In particular, it is shown that the proposed metric is capable of distinguishing diverse architecture classes. The GPUs showed to be the least flexible, which seems intuitive as their performance is heavily impacted by the amount of parallelism present in applications. Most flexible are the FPGAs, but interestingly only when high-level synthesis was not optimising. This and the other measurements also align with

[3]preliminary, since the results for this class of architectures is based on simulation and not measurements on real hardware

the idea that high flexibility is on tension with high performance, especially when not all applications profit from the applied optimisation. A similar conclusion can be drawn for the relation between energy efficiency.

Apart from the 25 COTS platforms, 40 ASIPs were used to investigate the relation between flexibility and customisation. Interestingly the results show that flexibility is a property that can be improved by adding more resources, similar to performance for example. When the available resources are scarce, flexibility is typically low as only some benchmarks benefit from added compute capabilities. With more resources available all applications in the benchmark set can be accelerated, often leading to better flexibility.

Furthermore an extensive discussion is provided on the state of the art, the proposed flexibility metric, and several alternative choices that could be made when moving the field forward. For instance, it is shown that the impact of the proposed intrinsic workload normalisation compared to normalising by a standard RISC is fairly limited. While the theoretical case for intrinsic workload is arguably stronger, using a RISC as normalisation may be a more practical way to move forward. The inclusion of loads and stores did not impact the flexibility significantly for the selected benchmarks, although for more complicated memory systems it most likely is interesting to consider them. In particular for applications that have a lot of data movement such as matrix transpose, accounting for loads and stores is expected to have a significant impact.

Finally the proposed flexibility metric is compared in depth to the two alternative definitions found in literature. In case of VersaBench versatility it is shown that in fact performance is measured, and not a orthogonal property. For versatility as defined by Van Berkel [77], it is argued that it measures a slightly different, yet related property. Instead of the direct flexibility, it measures how efficiently a computer architect managed to encode the workload of the application domain. Given these observations, it is concluded that the proposed flexibility metric has its own unique merits which warrant its introduction. Furthermore, it aligns with several key notions of flexibility that seem to be shared by a majority of the community, and as such serves as a good starting point in defining a commonly accepted definition of compute system flexibility.

Overall this chapter provides a survey of the current situation, a starting point in assessing processor flexibility in a quantitative manner, and lays the foundation for a broader discussion in the computer architecture and processor design community.

Chapter 7

Conclusions & Future Work

This chapter summarizes the main conclusions of the preceding chapters and their contributions towards improving both the compute and data efficiency of flexible architecture (section 7.1). Despite spanning several chapters and representing many years of research, the contributions encapsulated in this thesis are only a small step in the continuously evolving field of computer architecture. And although with this chapter this thesis has reached its end, by no means is the journey of improving compute machinery complete. Many avenues for further improvement remain unexplored. Section 7.2 lists several of these avenues that we consider interesting directions for future research.

7.1 Conclusions

The exponential scaling of compute power over the past decades has increasingly digitized the world to a point where almost all aspects of modern life depend on the availability of compute resources. It has firmly increased humanity's grip on its environment by enabling advanced control techniques, automation of repetitive tasks allowing people to spend their time solving more interesting issues, and the design of systems with a complexity and scale unimaginable without computers. This journey is far from over however, as advancements in the field continue to enable more complex applications. As detailed in chapter 1 convolutional neural networks (CNNs) are exemplary for such complex applications, which have entered the realm of feasible solutions to complex problems only as compute resources caught up with the theory behind these algorithms. Therefore it remains important to keep improving compute and data complexity of flexible machines.

In this thesis the state of the art is advanced in several ways. The contributions are logically divided into three categories; improving *compute efficiency*, improving *data efficiency*, and formally *defining compute flexibility*.

In chapter 2 a low-energy wide-single instruction multiple data (SIMD) architecture with explicit datapath is proposed, which through the use of a control processor (CP) and processing element (PE)-array exploits both data-level parallelism (DLP) and instruction-level parallelism (ILP). It is shown that the application of an explicit datapath reduces the register file (RF) accesses by 64 % on average for a 128 PE instance. This reduces the total energy dissipation compared to a version with implicitly bypassed datapath by 27.5 % on average, demonstrating the effectiveness of explicit datapath techniques in wide-SIMD architectures. Furthermore it is shown that a 128 PE instance of the proposed architecture improves the energy efficiency by 48.3 % on average compared to a single issue reduced instruction set computer (RISC) machine. This result is achieved without voltage-frequency scaling, which could easily be considered as the 128 PE SIMD additionally improves performance by a factor 206× on average. In this way the compute energy-efficiency could be improved even further.

Chapter 3 introduces two reduction algorithms which are optimized for the highly scalable, minimal interconnect applied in the SIMD proposed in chapter 2. It is shown that the algorithms are much more effective than a straightforward approach and can even compete with dedicated hardware solutions. This enables an efficient, programmable reduction operation, of which the runtime is independent of the specific reduction operator. In accordance with the flexibility metric proposed in chapter 6 this approach is therefore much more flexible than fixed function hardware. Since CNN applications heavily rely on reduction, the method introduced here can be used to both efficiently and flexibly support CNNs on the architecture proposed in chapter 2

Apart from reduction, multiplication is a cornerstone operation in neural network evaluation, as well as many other compute intensive tasks. The datawidth-aware techniques introduced in chapter 4 address exactly the compute efficiency of this all-important operator by exploiting that many operations in real-world applications do not utilize the full bit-width of the datapath. The best evaluated datawidth-aware multiplier design improves the energy efficiency of 32-bit × 32-bit multiplication by 38 % on average compared to a baseline multiplier. This efficiency is reached despite the overhead introduced by data-format conversion incurs a 25 % energy penalty, indicating the potential of this technique is even higher if the number of conversions can be limited, or more intelligently integrated into the multiplication logic.

The wide-SIMD, reduction algorithms, and multiplier optimizations introduced in chapters 2, 3, and 4 respectively all target improving the compute efficiency.

Another major source of energy consumption in modern technology nodes is accessing data in memory, which is inefficient due to the relatively poor scaling of memory compared to logic, a phenomenon known as the memory wall [177]. The memory wall also posses a risk for the efficient execution of data-intensive CNN applications. This is addressed in chapter 5, which introduces a model for selecting CNN execution schedules which minimize memory accesses by exploiting the abundant data-reuse present in convolutional neural networks. The execution schedules captured by this generic model include tiling, recomputation, and crucially loop fusion. The proposed model and accompanying open source tool [154] are shown to yield several Pareto optimal schedules missed by existing work. An exploration of the energy consumption for real-world networks suggests that a multi-level memory hierarchy is critical to effectively exploit the proposed schedules. Combined with an appropriate memory system however, the schedules found through the introduced framework demonstrate the potential to significantly improve the data efficiency of CNN execution.

It is commonly understood that generality of a compute platform can be traded for improvements in performance and/or energy efficiency. Over-specialisation of a platform, however, reduces its efficiency as applications change. In particular for neural network, who's underlying algorithms change rapidly, this poses as risk. Therefore it is desirable to design a compute system that is "flexible enough" to deal with potential application changes in the future. The lack of a formal definition of compute flexibility makes it difficult to assess this system property, and the impact of various architectural changes on it. In chapter 6 both an initial qualitative and quantitative definition of flexibility are proposed. These definitions enable objective measurement of system flexibility, and have been evaluated for 25 platforms over 14 benchmarks. General trends in these measurements appear to align with common notions surrounding flexibility, supporting the validity of the proposed definitions. The hope is that these initial definitions spark a wider discussion within the compute architecture community, ultimately leading to a better understanding of the concept of flexibility, and the design of flexible systems.

7.2 Future Work

As the title of this thesis suggests, advancing the efficiency of compute architectures is a continuous process which by no means is concluded by this thesis. Despite making several contributions to the field, as outlined in the preceding

section, many opportunities (fortunately) remain to improve the compute and data efficiency of future flexible architectures. The following topics in particular appear to be promising directions based on observations made during the work captured in this thesis:

- The wide-SIMD with explicit datapath introduced in chapter 2 provides a solid foundation, and could easily be improved with several extensions. In layout, to name an interesting example, the required clock-tree to supply all PEs with a synchronised clock could be a bottleneck, and require large energy-hungry cells to meet timing constraints. The use of only local wires to connect neighbouring PEs however lends itself perfectly for a mesochronous [96], i.e., globally asynchronous, locally synchronous, implementation. This would however require some (re)consideration of the interface towards data and instruction memory, and the CP broadcast signal.

- Additionally, layout-aware design could be used to introduce extra connections between PEs which are physically short, but logically long. I.e., if PEs are assumed to be laid out on chip in a kind of snake pattern, short wires could be introduced between PEs in parallel sections of the snake. Since these wires are physically short they do not impact scaling, although further research is required on how to efficiently exploit these irregular connections at the application/code-generation level.

- Another interesting topic is the addition of a branch predictor to the wide-SIMD. With its centralized control flow, a branch predictor is relatively cheap in a wide-SIMD while the gains can be significant. However, dynamic branch prediction is complex when combined with an explicit datapath which makes this an interesting challenge.

- The most energy-efficient datawidth-aware multiplier introduced in chapter 4 uses explicit conversions between the two's complement and sign magnitude data formats. These conversions can likely be integrated with the multiplication logic itself to reduce the area and energy overhead. Potentially it could also be interesting to allow both formats to exist throughout the datapath, and let a compiler explicitly schedule the conversions only when needed to further reduce the energy penalty.

- The CNN scheduling framework presented in chapter 5 covers a fairly wide range of code transformations. However, the supported layer types can still be extended significantly, for example by considering recurrent

layer types and residual connections. In particular for loop fusion, it could pay off to use a dataflow representation to capture the dependencies between operations, and compute buffer sizes based on standard dataflow techniques.

- Apart from increased layer support, it is also interesting to expand the CNN scheduling framework with support for multi-level memory hierarchies by introducing multiple store and compute levels. As the results presented in section 5.7 show, a multi-level memory hierarchy is essential to benefit from layer fusion techniques. The addition of a compute and store level per memory level would enable the exploration of schedules for machines with multiple levels of scratchpad memories.
- Finally, the flexibility metric proposed in chapter 6 should be considered only as a starting point. Research into different definitions and methods of measurement would be highly interesting, with the final goal to reach a consensus on a definition within the community. A multitude of decisions that can be reconsidered regarding the definition of flexibility have already been summarized in section 6.8.

Appendix A

SIMD Instruction Set

The table below contains the instruction set architecture (ISA) of the wide-SIMD presented in chapter 2. Note that the CP and PE share this ISA, with the exception of branch and jump instructions which are only available to the CP.

Instr.	Description	Operation
ADD	add signed	`rD = rA + rB`
AND	bitwise and	`rD = rA & rB`
CMOV	conditional move	`rD = flag ? rA : rB`
MUL	multiply signed	`rD = rA * rB`
MULU	multiply unsigned	`rD = rA * rB`
OR	bitwise or	`rD = rA or rB`
ROR	rotate register right	`rD[(N-1-rB[4:0]):0] = rA[N:rB[4:0]]` `rD[(N-1):(N-rB[4:0])] = rA[(rB[4:0]-1):0]`
SFEQ	set flag if equal	`flag = rA == rB`
SFGES	set flag if greater or equal, signed	`flag = rA >= rB`
SFGEU	set flag if greater or equal, unsigned	`flag = rA >= rB`
SFGTS	set flag if greater, signed	`flag = rA > rB`
SFGTU	set flag if greater, unsigned	`flag = rA > rB`
SFLES	set flag if less or equal, signed	`flag = rA <= rB`
SFLEU	set flag if less or equal, unsigned	`flag = rA <= rB`
SFLTS	set flag if less, signed	`flag = rA < rB`
SFLTU	set flag if less, unsigned	`flag = rA < rB`
SFNE	set flag if not equal	`flag = rA != rB`
SLL	shift left logical	`rD[(N-1):rB[4:0]] = rA[(N-1-rB[4:0]):0]` `rD[(rB[4:0]-1):0] = 0`

SRA	shift right arithmetic	rD[(N-1-rB[4:0]):0] = rA[(N-1):rB[4:0]] rD[(N-1):(N-rB[4:0])] = rA[N-1]
SRL	shift right logical	rD[(N-1-rB[4:0]):0] = rA[(N-1):rB[4:0]] rD[(N-1):(N-rB[4:0])] = 0
SUB	subtract signed	rD = rA - rB
XOR	bitwise xor	rD = rA xor rB
ADDI	add immediate signed	rD = rA + sign-ext(imm)
ANDI	and immediate unsigned	rD = rA & zero-ext(imm)
LWZ	low word	addr = rA + sign-ext(imm) rD = mem[addr]
MULI	multiply immediate signed	rD = rA * sign-ext(imm)
ORI	or immediate unsigned	rD = rA or zero-ext(imm)
SFEQI	set flag if equal immediate	flag = rA == sign-ext(imm)
SFGESI	set flag if greater or equal immediate, signed	flag = rA >= sign-ext(imm)
SFGEU	set flag if greater or equal immediate, unsigned	flag = rA >= sign-ext(imm)
SFGTS	set flag if greater immediate, signed	flag = rA > sign-ext(imm)
SFGTU	set flag if greater immediate, unsigned	flag = rA > sign-ext(imm)
SFLES	set flag if less or equal immediate, signed	flag = rA <= sign-ext(imm)
SFLEU	set flag if less or equal immediate, unsigned	flag = rA <= sign-ext(imm)
SFLTS	set flag if less immediate, signed	flag = rA < sign-ext(imm)
SFLTU	set flag if less immediate, unsigned	flag = rA < sign-ext(imm)

SFNE	set flag if not equal immediate signed	flag = rA != sign-ext(imm)
SW	store word	addr = rA + sign-ext(imm) mem[addr] = rB
XORI	xor immediate signed	rD = rA or sign-ext(imm)
BF	branch if flag	addr = branch_pc + sign-ext(imm«2) pc = addr, if flag == 1
BNF	branch if not flag	addr = branch_pc + sign-ext(imm«2) pc = addr, if flag == 0
J	jump	pc = branch_pc + sign-ext(imm«2)
JAL	jump and link	pc = branch_pc + sign-ext(imm«2) LR = branch_pc + 8
JALR	jump and link register	pc = rB LR = branch_pc + 8
JR	jump register	pc = rB
NOP	nop	
SLLI	shift left logical immediate	rD[(N-1):imm] = rA[(N-1-imm):0] rD[(rB[4:0]-1):0] = 0
SRAI	shift right arithmetic immediate	rD[(N-1-imm):0] = rA[(N-1):imm] rD[(N-1):(N-imm)] = rA[N-1]
SRLI	shift right logical immediate	rD[(N-1-imm):0] = rA[(N-1):imm] rD[(N-1):(N-imm)] = 0
ZIMM	zero extended long immediate (upper 18 bits for the immediate in the instruction following this ZIMM)	imm«8

SIMM	sign extended long immediate (upper 18 bits for the immediate in the instruction following this SIMM)	imm«8

Appendix B

Flexibility Related Lemmas

The eight lemmas that are instrumental to proving the properties of the flexibility metric proposed in section 6.3.2. Note that in this section the term "positive dataset" is understood to be a set of positive real numbers.

Lemma 1. *The geometric mean (GM) of a positive dataset* $\boldsymbol{X}$ *increases if an element* $x_i \in \boldsymbol{X}$ *increases.*

Proof Let $GM(\boldsymbol{X}) = (x_1 \cdot x_2 \cdots x_n)^{\frac{1}{n}}$ denote the geometric mean of positive dataset $\boldsymbol{X}$. Then for dataset $\boldsymbol{X}$ and its incremented version $\boldsymbol{X'}$:

$$GM(\boldsymbol{X})^n = x_1 \cdot x_2 \cdots x_n, \text{ and } GM(\boldsymbol{X'})^n = x_1 \cdot x_2 \cdots {x_k}' \cdots x_n,$$

where ${x_k}' = x_k + \epsilon$ with $\epsilon > 0$ It follows that:

$$\begin{aligned} GM(\boldsymbol{X'})^n - GM(\boldsymbol{X})^n &= (x_1 \cdot x_2 \cdots {x_k}' \cdots x_n) - (x_1 \cdot x_2 \cdots x_k \cdots x_n) \\ &= x_1 \cdot x_2 \cdots ({x_k}' - x_k) \cdots x_n > 0 \end{aligned}$$

And since $GM(\boldsymbol{X})^n$ is a positive, monotonically increasing function for positive dataset $\boldsymbol{X}$ and $n = |\boldsymbol{X}|$, it follows that $GM(\boldsymbol{X'}) > GM(\boldsymbol{X})$.

Lemma 2. *The geometric standard deviation (GSD) of a positive dataset* $\boldsymbol{X}$ *can either increase or decrease when an element of* $\boldsymbol{X}$ *increases.*

Proof Let $\boldsymbol{X} = [1, 3, 2, 2]$ be the original positive dataset, and $\boldsymbol{X'} = [2, 3, 2, 2]$, $\boldsymbol{X''} = [10, 3, 2, 2]$ be two datasets after increasing the first element of $\boldsymbol{X}$.

$$GSD(\boldsymbol{X}) = GSD([1, 3, 2, 2]) \approx 1.48$$

$$GSD(\boldsymbol{X'}) = GSD([2, 3, 2, 2]) \approx 1.19$$

$$GSD(\boldsymbol{X''}) = GSD([10, 3, 2, 2]) \approx 1.93$$

Hence, an increase of an element in $\boldsymbol{X}$ can either increase or decrease the geometric standard deviation of $\boldsymbol{X}$.

Lemma 3. *The geometric standard deviation (GSD) is invariant to multiplicative scaling, i.e., $GSD(s \cdot \boldsymbol{X}) = GSD(\boldsymbol{X})$, where $\boldsymbol{X}$ is a positive dataset and 's' is a positive constant.*

Proof

$$GM(s \cdot \boldsymbol{X}) = \left(\prod_{i=1}^{n} s \cdot x_i \right)^{\frac{1}{n}} = \left(s^n \prod_{i=1}^{n} x_i \right)^{\frac{1}{n}} = s \cdot GM(\boldsymbol{X})$$

$$\begin{aligned} GSD(s \cdot \boldsymbol{X}) &= \exp\left(\sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(\ln \frac{s \cdot x_i}{GM(s \cdot \boldsymbol{X})} \right)^2} \right) \\ &= \exp\left(\sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(\ln \frac{x_i}{GM(\boldsymbol{X})} \right)^2} \right) = GSD(\boldsymbol{X}) \end{aligned}$$

Lemma 4. *The geometric mean (GM) of a dataset $\boldsymbol{X}$ normalised to dataset $\boldsymbol{B}$ is equal to the ratio of the GMs of $\boldsymbol{X}$, and $\boldsymbol{B}$, i.e., $GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right) = \frac{GM(\boldsymbol{X})}{GM(\boldsymbol{B})}$, where $\boldsymbol{X} = [x_1, x_2, \ldots, x_n]$ and $\boldsymbol{B} = [b_1, b_2, \ldots, b_n]$ are positive datasets.*

Proof

$$\frac{GM(\boldsymbol{X})}{GM(\boldsymbol{B})} = \frac{\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}}}{\left(\prod_{i=1}^{n} b_i\right)^{\frac{1}{n}}} = \left(\frac{\prod_{i=1}^{n} x_i}{\prod_{i=1}^{n} b_i} \right)^{\frac{1}{n}} = \left(\prod_{i=1}^{n} \frac{x_i}{b_i} \right)^{\frac{1}{n}} = GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right)$$

Lemma 5. *The geometric mean of dataset $\boldsymbol{X}$ normalised to dataset $\boldsymbol{B}$ is equal to the reciprocal of the geometric mean of $\boldsymbol{B}$ normalised to $\boldsymbol{X}$, i.e., $GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right) = GM\left(\frac{\boldsymbol{B}}{\boldsymbol{X}}\right)^{-1}$*

Proof

$$GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right) \overset{\text{lemma 4}}{=\!=} \frac{GM(\boldsymbol{X})}{GM(\boldsymbol{B})} = \left(\frac{GM(\boldsymbol{B})}{GM(\boldsymbol{X})}\right)^{-1} \overset{\text{lemma 4}}{=\!=} GM\left(\frac{\boldsymbol{B}}{\boldsymbol{X}}\right)^{-1}$$

Lemma 6. *The ratio of the geometric means (GMs) of different normalised dataset is the same as the ratio of the GMs of the original datasets, i.e., $\frac{GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right)}{GM\left(\frac{\boldsymbol{Y}}{\boldsymbol{B}}\right)} = \frac{GM(\boldsymbol{X})}{GM(\boldsymbol{Y})}$, where $\boldsymbol{X}$, $\boldsymbol{Y}$, and $\boldsymbol{B}$ are positive datasets.*

Proof

$$\frac{GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right)}{GM\left(\frac{\boldsymbol{Y}}{\boldsymbol{B}}\right)} \overset{\text{lemma 4}}{=\!=} \frac{\frac{GM(\boldsymbol{X})}{GM(\boldsymbol{B})}}{\frac{GM(\boldsymbol{Y})}{GM(\boldsymbol{B})}} = \frac{GM(\boldsymbol{X})}{GM(\boldsymbol{Y})}$$

Lemma 7. *The geometric standard deviation (GSD) of a normalised dataset is equal to the GSD of the reciprocal of that normalised dataset, i.e.,* $GSD\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right) = GSD\left(\frac{\boldsymbol{B}}{\boldsymbol{X}}\right)$*, where* $\boldsymbol{X}$ *and* $\boldsymbol{B}$ *are positive datasets.*

Proof

$$\begin{aligned}
GSD\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right) &= \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{\frac{x_i}{b_i}}{GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right)}\right)^2}\right) \\
&= \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln x_i - \ln\left(b_i \cdot GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right)\right)\right)^2}\right) \\
&= \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\left(b_i \cdot GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right)\right) - \ln x_i\right)^2}\right) \\
&= \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{b_i \cdot GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right)}{x_i}\right)^2}\right) \\
&\overset{\text{lemma 5}}{=\!=} \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{\frac{b_i}{x_i}}{GM\left(\frac{\boldsymbol{B}}{\boldsymbol{X}}\right)}\right)^2}\right) \\
&= GSD\left(\frac{\boldsymbol{B}}{\boldsymbol{X}}\right)
\end{aligned}$$

Lemma 8. *The geometric standard deviation (GSD) of a normalised dataset is always greater than or equal to the ratio of the GSDs of the original dataset and the baseline, i.e.,* $GSD\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right) \geq \frac{GSD(\boldsymbol{X})}{GSD(\boldsymbol{B})}$*, where* $\boldsymbol{X}$ *and* $\boldsymbol{B}$ *are real positive datasets.*

Proof First rewrite $GSD\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right)$ and $\frac{GSD(\boldsymbol{X})}{GSD(\boldsymbol{B})}$:

$$\begin{aligned} GSD\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right) &= \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{\frac{x_i}{b_i}}{GM\left(\frac{\boldsymbol{X}}{\boldsymbol{B}}\right)}\right)^2}\right) \\ &\overset{\text{lemma 4}}{=\!=} \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{\frac{x_i}{b_i}}{\frac{GM(\boldsymbol{X})}{GM(\boldsymbol{B})}}\right)^2}\right) \\ &= \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\boldsymbol{X})} - \ln\frac{b_i}{GM(\boldsymbol{B})}\right)^2}\right) \end{aligned}$$

$$\begin{aligned} \frac{GSD(\boldsymbol{X})}{GSD(\boldsymbol{B})} &= \frac{\exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\boldsymbol{X})}\right)^2}\right)}{\exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{b_i}{GM(\boldsymbol{B})}\right)^2}\right)} \\ &= \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\boldsymbol{X})}\right)^2} - \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{b_i}{GM(\boldsymbol{B})}\right)^2}\right) \end{aligned}$$

Since $\exp(y)$ is a positive monotonic function, it is sufficient to prove:

$$\begin{aligned} &\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\boldsymbol{X})} - \ln\frac{b_i}{GM(\boldsymbol{B})}\right)^2} \geq \\ &\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\boldsymbol{X})}\right)^2} - \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{b_i}{GM(\boldsymbol{B})}\right)^2} \end{aligned}$$

Substituting $\boldsymbol{U} = \ln\frac{x_i}{GM(\boldsymbol{X})}$, and $\boldsymbol{V} = \ln\frac{b_i}{GM(\boldsymbol{B})}$ yields:

$$||\boldsymbol{U} - \boldsymbol{V}|| \geq ||\boldsymbol{U}|| - ||\boldsymbol{V}||$$

Which holds according to the reverse triangle inequality.

Abbreviations

2C	two's complement
ALU	arithmetic logic unit
AM	arithmetic mean
ANN	artificial neural network
ASD	arithmetic standard deviation
ASIC	application-specific integrated circuit
ASIP	application-specific instruction-set processor
CH	cumulative histogram
CIA	cumulative intensive area
CMOS	complementary metal oxide semiconductor
CNN	convolutional neural network
COTS	commercial of the shelf
CP	control processor
CPU	central processing unit
DLP	data-level parallelism
DMEM	data memory
DNN	deep neural network
DRAM	dynamic random-access memory
DSE	design space exploration
DSP	digital signal processor
DVFS	dynamic voltage frequency scaling
ED	energy-delay product
EDP	energy-delay-power product
EW	effective data-width
EX	execution stage
FFOS	fast focus on structures
FPGA	field-programmable gate array
FU	functional unit

GM	geometric mean
GPGPU	general purpose computing on graphics processing units
GPP	general purpose processor
GPU	graphics processing unit
GSD	geometric standard deviation
HDL	hardware description language
HLS	high-level synthesis
ID	instruction decode
IF	instruction fetch
ILP	instruction-level parallelism
ILSVRC	ImageNet large scale visual recognition challenge
IMEM	instruction memory
IR	intermediate representation
ISA	instruction set architecture
LIW	long instruction word
LSU	load store unit
LUT	lookup table
MAC	multiply accumulate
MAD	median absolute deviation
MEW	maximum effective data-width
MSB	most significant bit
MUL	multiplier unit
NOP	no-operation
OLED	organic light-emitting diode
OS	operating system
PE	processing element
RF	register file
RISC	reduced instruction set computer
RTL	register-transfer level

SIMD	single instruction multiple data
SM	sign magnitude
SRAM	static random-access memory
TDP	thermal design power
TTA	transport triggered architecture
VLIW	very long instruction word
WB	write back stage

Bibliography

[1] *128-tap FIR bandpass filter,* $300 - 5kHz$. http://t-filter.appspot.com/fir/index.html.

[2] A.A. Abbo, R.P. Kleihorst, V. Choudhary, L. Sevat, P. Wielage, S. Mouy, B. Vermeulen, and M. Heijligers. "Xetal-II: A 107 GOPS, 600 mW Massively Parallel Processor for Video Scene Analysis". In: *IEEE Journal of Solid-State Circuits (JSSC)* 43.1 (Jan. 2008), pp. 192–201. ISSN: 0018-9200. DOI: 10.1109/JSSC.2007.909328.

[3] David Abdurachmanov, Peter Elmer, Giulio Eulisse, and Shahzad Muzaffar. "Initial explorations of ARM processors for scientific computing". In: *Journal of Physics: Conference Series* 523.1 (2014), p. 012009.

[4] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. "State-of-the-art in artificial neural network applications: A survey". In: *Heliyon* 4.11 (2018), e00938. ISSN: 2405-8440. DOI: https://doi.org/10.1016/j.heliyon.2018.e00938. URL: https://www.sciencedirect.com/science/article/pii/S2405844018332067.

[5] Michaël Adriaansen, Mark Wijtvliet, Roel Jordans, Luc Waeijen, and Henk Corporaal. "Code Generation for Reconfigurable Explicit Datapath Architectures with LLVM". In: *2016 Euromicro Conference on Digital System Design (DSD)*. Aug. 2016, pp. 30–37. DOI: 10.1109/DSD.2016.88.

[6] Taekyoon Ahn and Kiyoung Choi. "Dynamic operand interchange for low power". In: *Electronics Letters* 33.25 (Dec. 1997), pp. 2118–2120. ISSN: 0013-5194. DOI: 10.1049/el:19971440.

[7] M. Alwani, H. Chen, M. Ferdman, and P. Milder. "Fused-layer CNN accelerators". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783725.

[8] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18 x2013;20), AFIPS Press, Reston, Va., 1967, pp. 483 x2013;485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California". In: *Solid-State Circuits Society Newsletter, IEEE* 12.3 (2007), pp. 19–20. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2007.4785615.

[9] Anandtech. *The Haswell Review: Intel Core i7-4770K & i5-4670K Tested*. https://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested, accessed 2021-3-30. 2013.

[10] Chris Angelini. *GeForce GTX Titan X Review: Can One GPU Handle 4K?* https://www.tomshardware.com/reviews/nvidia-geforce-gtx-titan-x-gm200-maxwell,4091.html, accessed on 2021-3-30. 2015.

[11] Y. Arai, T. Agui, and M. Nakajima. "A fast dct-sq scheme for images". In: *Ieice Transactions* (1988).

[12] M. Arnold and H. Corporaal. "Automatic detection of recurring operation patterns." English. In: *Proceedings of the Seventh International Workshop on Hardware/Software Codesign : (CODES '99), May 3 - 5, 1999, Rome, Italy*. United States: Association for Computing Machinery, Inc, 1999, pp. 22–26. ISBN: 1-58113-132-1.

[13] Lennart Bamberg, Arash Pourtaherian, Luc Waeijen, Anupam Chahar, and Orlando Moreira. *Synapse Compression for Event-Based Convolutional Neural Network Accelerators*. 2021. arXiv: 2112.07019 [cs.AR].

[14] C. R. Baugh and B. A. Wooley. "A Two's Complement Parallel Array Multiplication Algorithm". In: *IEEE Transactions on Computers* C-22.12 (Dec. 1973), pp. 1045–1047. ISSN: 0018-9340. DOI: 10.1109/T-C.1973.223648.

[15] M. Bhardwaj, R. Min, and A. P. Chandrakasan. "Quantifying and Enhancing Power Awareness of VLSI Systems". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9.6 (Dec. 2001), pp. 757–772. ISSN: 1063-8210. DOI: 10.1109/92.974890.

[16] Jarno Brils, Luc Waeijen, and Arash Pourtaherian. "How to Exploit Sparsity in RNNs on Event-Driven Architectures". In: *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '21. Eindhoven, Netherlands: Association for Computing Machinery, 2021, pp. 17–22. ISBN: 9781450391665. DOI: 10.1145/3493229.3493302. URL: https://doi.org/10.1145/3493229.3493302.

[17] D. Brooks and M. Martonosi. "Dynamically exploiting narrow width operands to improve processor power and performance". In: *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. Jan. 1999, pp. 13–22. DOI: 10.1109/HPCA.1999.744314.

[18] David Brooks and Margaret Martonosi. "Value-based Clock Gating and Operation Packing: Dynamic Strategies for Improving Processor Power and Performance". In: *ACM Trans. Comput. Syst.* 18.2 (May 2000), pp. 89–126. ISSN: 0734-2071. DOI: 10.1145/350853.350856.

[19] David Buchfuhrer and Christopher Umans. “The Complexity of Boolean Formula Minimization”. In: vol. 77. July 2008, pp. 24–35.

[20] cadence. *Encounter(R) RTL Compiler*. Version v11.20. URL: https://www.csee.umbc.edu/~tinoosh/cmpe641/tutorials/rc/rc_commandref.pdf.

[21] Hyunman Chang, Soohwan Ong, Changhee Lee, M.H. Sunwoo, and Taihoon Cho. “A general purpose SliM-II image processor”. In: *Computer Architecture for Machine Perception, 1997. CAMP 97. Proceedings. 1997 Fourth IEEE International Workshop on*. 1997, pp. 253–259. DOI: 10.1109/CAMP.1997.632034.

[22] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks”. In: *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 262–263.

[23] François Chollet et al. *Keras*. 2015. URL: https://keras.io.

[24] G. Chryssolouris. “Flexibility and Its Measurement”. In: *CIRP Annals* 45.2 (1996), pp. 581–587. ISSN: 0007-8506.

[25] Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen Schmidhuber. “Flexible, High Performance Convolutional Neural Networks for Image Classification”. In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*. IJCAI’11. Barcelona, Catalonia, Spain: AAAI Press, 2011, pp. 1237–1242. ISBN: 9781577355144.

[26] Michael Cloutier, Chad Paradis, and Vincent Weaver. “A Raspberry Pi Cluster Instrumented for Fine-Grained Power Measurement”. In: *Electronics* 5 (Sept. 2016), p. 61.

[27] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule. “Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications”. In: *IEEE Micro* 34.2 (Mar. 2014). ISSN: 0272-1732.

[28] *Color Conversion Application Note version 1.4*. http://www.stretchinc.com/_files/Color_Conversion_App_Note_v1_4.pdf.

[29] Henk Corporaal. *Microprocessor Architectures: From VLIW to TTA*. Wiley, 1998. ISBN: 047197157X.

[30] R. Damodaran et al. “A 1.25GHz 0.8W C66x DSP Core in 40nm CMOS”. In: *2012 25th International Conference on VLSI Design*. Hyderabad, India: IEEE, Jan. 2012, pp. 286–291.

[31] Delft University of Technology. *MOVE project*. http://openasip.org/move/DelftMoveSite/MOVE/documents.html. [Online; accessed 01-05-2021].

[32] Robert Fasthuber, Francky Catthoor, Praveen Raghavan, and Frederik Naessens. *Energy-Efficient Communication Processors: Design and Implementation for Emerging Wireless Systems*. Springer Publishing Company, Incorporated, 2013. ISBN: 9781461449911.

[33] Joseph A. Fisher, Paolo Faraboschi, and Giuseppe Desoli. "Custom-Fit Processors: Letting Applications Define Architectures". In: *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 29. Paris, France: IEEE Computer Society, 1996, pp. 324–335. ISBN: 0818676418.

[34] Philip J. Fleming and John J. Wallace. "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results". In: *Commun. ACM* 29.3 (Mar. 1986), pp. 218–221. ISSN: 0001-0782.

[35] R. Frijns, H. Fatemi, B. Mesman, and H. Corporaal. "DC-SIMD : Dynamic Communication for SIMD Processors". In: *Proceedings of International Symposium on Parallel and Distributed Processing (IPDPS)*. 2008, pp. 1–10. DOI: 10.1109/IPDPS.2008.4536274.

[36] M. Fujino and V. G. Moshnyaga. "Dynamic operand transformation for low-power multiplier-accumulator design". In: *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*. Vol. 5. May 2003, V-345-V–348 vol.5. DOI: 10.1109/ISCAS.2003.1206276.

[37] Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Davide Rossi, and Luca Benini. "XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2020, pp. 186–191. DOI: 10.23919/DATE48585.2020.9116529.

[38] Tong Geng, Luc Waeijen, Maurice Peemen, Henk Corporaal, and Yifan He. "MacSim: A MAC-Enabled High-Performance Low-Power SIMD Architecture". In: *2016 Euromicro Conference on Digital System Design (DSD)*. Aug. 2016, pp. 160–167. DOI: 10.1109/DSD.2016.27.

[39] N. Goel, A. Kumar, and P.R. Panda. "Power Reduction in VLIW Processor with Compiler Driven Bypass Network". In: *Proceedings of the 20th International Conference on VLSI Design (VLSID)*. Jan. 2007, pp. 233–238. DOI: 10.1109/VLSID.2007.127.

[40] K. Goetschalckx and M. Verhelst. "Breaking High-Resolution CNN Bandwidth Barriers With Enhanced Depth-First Execution". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 323–331.

[41] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[42] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger. Vol. 27. Curran Associates, Inc., 2014.

[43] M. K. Gowan, L. L. Biro, and D. B. Jackson. "Power considerations in the design of the Alpha 21264 microprocessor". In: *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*. June 1998, pp. 726–731. DOI: 10.1145/277044.277226.

[44] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. "Auto-tuning a high-level language targeted to GPU codes". In: *2012 Innovative Parallel Computing (InPar)*. May 2012, pp. 1–10.

[45] Xuan Guan and Yunsi Fei. "Reducing power consumption of embedded processors through register file partitioning and compiler support". In: *Proceedings of International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. 2008, pp. 269–274. DOI: 10.1109/ASAP.2008.4580190.

[46] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: http://doi.acm.org/10.1145/42411.42415.

[47] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. Dec. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[48] *H.264 reference implementation version JM18.6*. http://iphome.hhi.de/suehring/tml/.

[49] Rehan Hameed. *Balancing Efficiency and Flexibility In Specialized Computing*. PhD thesis. 2013.

[50] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. "Understanding Sources of Inefficiency in General-Purpose Chips". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: Association for Computing Machinery, 2010, pp. 37–47. ISBN: 9781450300537. DOI: 10.1145/1815961.1815968. URL: https://doi.org/10.1145/1815961.1815968.

[51] Kyungtae Han, B. L. Evans, and E. E. Swartzlander. "Low-Power Multipliers with Data Wordlength Reduction". In: *Conference Record of the*

Thirty-Ninth Asilomar Conference onSignals, Systems and Computers, 2005. Oct. 2005, pp. 1615–1619. DOI: 10.1109/ACSSC.2005.1600041.

[52] M. Hatamian and G. L. Cash. "A 70-MHz 8-bit × 8-bit Parallel Pipelined Multiplier in 2.5-μm CMOS". In: *IEEE Journal of Solid-State Circuits* 21.4 (Aug. 1986), pp. 505–513. ISSN: 0018-9200. DOI: 10.1109/JSSC.1986.1052564.

[53] Y. He, Z. Ye, D. She, R.S. Pieters, B. Mesman, and H. Corporaal. "1000 fps visual servoing on the reconfigurable wide SIMD processor". In: *Proceedings of the 16th Annual Conference of the Advanced School for Computing and Imaging (ASCI)*. 2010, pp. 302–309.

[54] Yifan He. *Low Power Architectures for Streaming Applications*. PhD Thesis, 2013.

[55] Yifan He, Maurice Peemen, Luc Waeijen, Erkan Diken, Mattia Fiumara, Gerard Rauwerda, Henk Corporaal, and Tong Geng. "A configurable SIMD architecture with explicit datapath for intelligent learning". In: *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. July 2016, pp. 156–163. DOI: 10.1109/SAMOS.2016.7818343.

[56] Yifan He, Yu Pu, Zhenyu Ye, S.M. Londono, R. Kleihorst, A.A. Abbo, and H. Corporaal. "Xetal-Pro: An ultra-low energy and high throughput SIMD processor". In: *Proceedings of the 47th Design Automation Conference (DAC)*. June 2010, pp. 543–548.

[57] Yifan He, Dongrui She, B. Mesman, and H. Corporaal. "MOVE-Pro: A low power and high code density TTA architecture". In: *Proceedings of the 11th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2011, pp. 294–301. DOI: 10.1109/SAMOS.2011.6045474.

[58] Yifan He, Zhenyu Ye, Dongrui She, Bart Mesman, and Henk Corporaal. "Feasibility analysis of ultra high frame rate visual servoing on FPGA and SIMD processor". In: *Proceedings of Advances Concepts for Intelligent Vision Systems (ACIVS)*. 2011, pp. 623–634.

[59] Yifan He, Zoran Zivkovic, Richard Kleihorst, Alexander Danilin, and Henk Corporaal. "Real-time implementations of Hough Transform on SIMD architecture". In: *Proceedings of the ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC)*. 2008, pp. 1–8.

[60] Yifan He, Zoran Zivkovic, Richard Kleihorst, Alexander Danilin, Henk Corporaal, and Bart Mesman. "Real-Time Hough Transform on 1-D SIMD Processors: Implementation and Architecture Exploration". In: *Proceed-*

ings of the International Conference Advanced Concepts for Intelligent Vision Systems (ACIVS). 2008, pp. 254–265.

[61] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[62] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. "A Fast Learning Algorithm for Deep Belief Nets". In: *Neural Comput.* 18.7 (July 2006), pp. 1527–1554. ISSN: 0899-7667. DOI: 10.1162/neco.2006.18.7.1527. URL: https://doi.org/10.1162/neco.2006.18.7.1527.

[63] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[64] J J Hopfield. "Neural networks and physical systems with emergent collective computational abilities". In: *Proceedings of the National Academy of Sciences* 79.8 (1982), pp. 2554–2558. ISSN: 0027-8424. DOI: 10.1073/pnas.79.8.2554. eprint: https://www.pnas.org/content/79/8/2554.full.pdf. URL: https://www.pnas.org/content/79/8/2554.

[65] Shihua Huang and Luc Waeijen. *Intrinsic WorkloadEstimator*. https://github.com/lwaeijen/WorkloadEstimator. 2021.

[66] Shihua Huang, Luc Waeijen, and Henk Corporaal. "How Flexible is Your Computing System?" In: *ACM Trans. Embed. Comput. Syst.* (Mar. 2022). ISSN: 1539-9087. DOI: 10.1145/3524861. URL: https://doi.org/10.1145/3524861.

[67] Amr Hussam Ibrahim, Mohamed Bakr Abdelhalim, Hanadi Hussein, and Ahmed Fahmy. *An Analysis of x86-64 Instruction Set for Optimization of System Softwares*. 2011.

[68] Jos IJzerman, Timo Viitanen, Pekka Jääskeläinen, Heikki Kultala, Lasse Lehtonen, Maurice Peemen, Henk Corporaal, and Jarmo Takala. "AivoTTA: An Energy Efficient Programmable Accelerator for CNN-Based Object Recognition". In: *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. SAMOS '18. Pythagorion, Greece: Association for Computing Machinery, 2018, pp. 28–37. ISBN: 9781450364942. DOI: 10.1145/3229631.3229637. URL: https://doi.org/10.1145/3229631.3229637.

[69] Texas Instruments. *Code Composer Studio*. Version V4. URL: https://software-dl.ti.com/ccs/esd/documents/ccs_downloads.html#code-composer-studio-version-4-downloads.

[70] Intel. https://ark.intel.com/#@Processors, accessed 2021-3-30.

[71] Intel. *Inside 6th GEN Intel Core: New Microarchitecture Code Named Skylake*. https://old.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.911-Skylake-Doweck-Intel_SK3-r13b.pdf, accessed 2021-3-30. 2016.

[72] Mike Jongen, Luc Waeijen, Roel Jordans, Lech Jozwiak, and Henk Corporaal. "Optimization through recomputation in the polyhedral model". English. In: *Eighth International Workshop on Polyhedral Compilation Techniques*. 8th International Workshop on Polyhedral Compilation Techniques (IMPACT 2018), January 23, 2018, Manchester, UK, IMPACT ; Conference date: 23-01-2018 Through 23-01-2018. Jan. 22, 2018. URL: http://impact.gforge.inria.fr/impact2018.

[73] *JPEG Encode from MiBench*. http://www.eecs.umich.edu/mibench/source.html.

[74] U.J. Kapasi, W.J. Dally, S. Rixner, J.D. Owens, and B. Khailany. "The Imagine Stream Processor". In: *Proceedings of International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*. 2002, pp. 282–288. DOI: 10.1109/ICCD.2002.1106783.

[75] Götz Kappen and Tobias Noll. "Application specific instruction processor based implementation of a GNSS receiver on an FPGA." In: Jan. 2006, pp. 58–63.

[76] Kingshuk Karuri and Rainer Leupers. *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*. Springer Publishing Company, Incorporated, 2014. ISBN: 9781493902309.

[77] Kees van Berkel. *Processor Versatility (Flexibility) - an attempt at definition and quantification*. MPSoC 2013, July 2013. URL: http://mpsoc-forum.org/archive/2013/slides/12-Van_Berkel.pdf (visited on 03/26/2021).

[78] W. Kellerer, A. Basta, P. Babarczi, A. Blenk, M. He, M. Klugel, and A. M. Alba. "How to Measure Network Flexibility? A Proposal for Evaluating Softwarized Networks". In: *IEEE Communications Magazine* (2018), pp. 2–8. ISSN: 0163-6804.

[79] T. B. L. Kirkwood. "Geometric Standard Deviation - Reply to Bohidar". In: *Drug Development and Industrial Pharmacy* 19.3 (1993), pp. 395–396.

[80] H. Kojima, D. J. Gorny, K. Nitta, and K. Sasaki. "Power analysis of a programmable DSP for architecture/program optimization". In: *1995 IEEE Symposium on Low Power Electronics. Digest of Technical Papers*. Oct. 1995, pp. 26–27. DOI: 10.1109/LPE.1995.485383.

[81] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings*

of the 25th International Conference on Neural Information Processing Systems - Volume 1. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105. URL: http://dl.acm.org/citation.cfm?id=2999134.2999257.

[82] S. r. Kuang and J. p. Wang. "Design of power-efficient pipelined truncated multipliers with various output precision". In: *IET Computers Digital Techniques* 1.2 (Mar. 2007), pp. 129–136. ISSN: 1751-8601. DOI: 10.1049/iet-cdt:20060156.

[83] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction". In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* Dec. 2003, pp. 81–92.

[84] S. Kyo and S. Okazaki. "IMAPCAR: A 100 GOPS in-vehicle vision processor based on 128 ring connected four-way VLIW processing elements". In: *Journal of Signal Processing Systems* (2008), pp. 1–12.

[85] Stanford Vision Lab. *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)*. https://www.image-net.org/challenges/LSVRC/index.php. Accessed 26-09-2021.

[86] *LAME MP3 encoder version 3.99*. http://sourceforge.net/projects/lame/files/lame/3.99/.

[87] R. Landauer. "Irreversibility and Heat Generation in the Computing Process". In: *IBM Journal of Research and Development* 5.3 (1961), pp. 183–191.

[88] E. Lannoye, D. Flynn, and M. O'Malley. "Evaluation of Power System Flexibility". In: *IEEE Transactions on Power Systems* 27.2 (May 2012), pp. 922–931. ISSN: 0885-8950.

[89] Y. Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Handwritten Digit Recognition with a Back-Propagation Network". In: *Proceedings of the 2nd International Conference on Neural Information Processing Systems*. NIPS'89. Cambridge, MA, USA: MIT Press, 1989, pp. 396–404.

[90] Dong -X. Li, Wei Zheng, and Ming Zhang. "Architecture Design for H.264/AVC Integer Motion Estimation with Minimum Memory Bandwidth". In: *Consumer Electronics, IEEE Transactions on* 53.3 (2007), pp. 1053–1060. ISSN: 0098-3063. DOI: 10.1109/TCE.2007.4341585.

[91] G. Li, F. Li, T. Zhao, and J. Cheng. "Block convolution: Towards memory-efficient inference of large-scale CNNs on FPGA". In: *2018 Design, Au-*

tomation & Test in Europe Conference & Exhibition (DATE). 2018, pp. 1163–1166.

[92] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li. "SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2018, pp. 343–348.

[93] Yuan Lin, Hyunseok Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. "SODA: A Low-power Architecture For Software Radio". In: *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*. 2006, pp. 89–101. DOI: 10.1109/ISCA.2006.37.

[94] Fei Liu, Yi Liang, and Lingze Wang. "A Survey of the Heterogeneous Computing Platform and Related Technologies". In: *DEStech Transactions on Engineering and Technology Research* (May 2017).

[95] Stef Louwers, Luc Waeijen, Mark Wijtvliet, Ruud Koolen, and Henk Corporaal. "Multi-granular Arithmetic in a Coarse-Grain Reconfigurable Architecture". In: *2016 Euromicro Conference on Digital System Design (DSD)*. Aug. 2016, pp. 599–606. DOI: 10.1109/DSD.2016.98.

[96] Daniele Ludovici, Alessandro Strano, Georgi Gaydadjiev, and Davide Bertozzi. "Mesochronous NoC technology for power-efficient GALS MP-SoCs". In: *INA-OCMC '11*. 2011.

[97] Pedro M.M. Pereira, Patrício Domingues, Nuno Rodrigues, Gabriel Falcao, and Sergio De Faria. "Assessing the Performance and Energy Usage of Multi-CPUs, Multi-Core and Many-Core Systems : The MMP Image Encoder Case Study". In: *International Journal of Distributed and Parallel systems* 7 (Sept. 2016), pp. 01–20.

[98] O. L. Macsorley. "High-Speed Arithmetic in Binary Computers". In: *Proceedings of the IRE* 49.1 (Jan. 1961), pp. 67–91. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1961.287779.

[99] Songping Mai, Chun Zhang, Yixin Zhao, Jun Chao, and Zhihua Wang. "An application-specific memory partitioning method for low power". In: *2007 7th International Conference on ASIC*. 2007, pp. 221–224. DOI: 10.1109/ICASIC.2007.4415607.

[100] Krishna T. Malladi, Frank A. Nothaft, Karthika Periyathambi, Benjamin C. Lee, Christos Kozyrakis, and Mark Horowitz. "Towards energy-proportional datacenter memory with mobile DRAM". In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 2012, pp. 37–48. DOI: 10.1109/ISCA.2012.6237004.

[101] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1, 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.

[102] Linyan Mei, Pouya Houshmand, Vikram Jain, Juan Sebastian P Giraldo, and Marian Verhelst. “ZigZag: A Memory-Centric Rapid DNN Accelerator Design Space Exploration Framework.” In: *CoRR (Computing Research Repository)* abs/2007.11360 (2020).

[103] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[104] Martinez MN and Bartholomew MJ. “What Does It Mean? A Review of Interpreting and Calculating Different Types of Means and Standard Deviations”. In: *Pharmaceutics* (Apr. 2017).

[105] Geoffrey Ndu. *Boosting Single Thread Performance in Mobile Processors using Reconfigurable Acceleration*. PhD thesis. Oct. 2012.

[106] Linda Null and Julia Lobur. *The Essentials of Computer Organization and Architecture*. 4th. USA, 2014. ISBN: 9781284045611.

[107] Nvidia. *GeForce Specifications*. https://www.nvidia.com/en-us/geforce/. accessed 2021-3-30.

[108] Nvidia. “Whitepaper NVIDIA Tegra K1: A New Era in Mobile Computing”. In: (Jan. 2014). URL: https://www.nvidia.com/content/PDF/tegra_white_papers/Tegra-K1-whitepaper-v1.0.pdf.

[109] nvidia. *CUDA Zone*. URL: https://developer.nvidia.com/cuda-zone.

[110] *OpenRISC*. http://opencores.org/or1k/Main_Page.

[111] Vishwamitra Oree and Sayed Z. Sayed Hassen. “A composite metric for assessing flexibility available in conventional generators of power systems”. In: *Applied Energy* 177 (2016), pp. 683–691. ISSN: 0306-2619.

[112] Nobuyuki Otsu. “A Threshold Selection Method from Gray-Level Histograms”. In: *Systems, Man and Cybernetics, IEEE Transactions on* 9.1 (1979), pp. 62–66. ISSN: 0018-9472. DOI: 10.1109/TSMC.1979.4310076.

[113] A.N. Page and A.S. Schwier. *Manual of Political Economy: By Vilfredo Pareto. Translated by Ann S. Schwier. Edited by Ann S. Schwier and Alfred N. Page*. MacMillan, 1972.

[114] Maurice Peemen, Bart Mesman, and Henk Corporaal. “Inter-tile Reuse Optimization Applied to Bandwidth Constrained Embedded Accelerators”. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. DATE ’15. Grenoble, France: EDA Consortium, 2015, pp. 169–174. ISBN: 978-3-9815370-4-8. URL: http://dl.acm.org/citation.cfm?id=2755753.2755790.

[115] S. Perri, M. Lanuzza, P. Corsonello, and G. Cocorullo. “SIMD 2D Convolver for Fast FPGA-based Image and Video Processors”. In: *Military Aerospace Programmalbe Logic Devices, 2003 (MAPLD'2003)*. 2003.

[116] A. Prengler and K. Adi. “A reconfigurable SIMD-MIMD processor architecture for embedded vision processing applications”. In: *SAE World Congress*. 2009, pp. 1–9.

[117] The Hamilton Project. *Cost of Computing Power Equal to an iPad2*. https://www.hamiltonproject.org/charts/cost_of_computing_power_equal_to_an_ipad2.

[118] Yu Pu, Yifan He, Zhenyu Ye, S.M. Londono, A.A. Abbo, R. Kleihorst, and H. Corporaal. “From Xetal-II to Xetal-Pro: On the Road Toward an Ultra low-Energy and High-Throughput SIMD Processor”. In: *IEEE Transactions on Circuits and Systems for Video Technology (TCAS-VT)* 21.4 (Apr. 2011), pp. 472–484. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2011.2125590.

[119] Qualcomm. *Hexagon SDK*. Version v3.5.3 Linux. URL: https://developer.qualcomm.com/downloads/hexagon-sdk-v354-linux.

[120] Qualcomm. “Hexagon Simulator User Guide”. In: (Dec. 2016).

[121] Qualcomm. “Qualcomm Hexagon DSP”. In: (Dec. 2017).

[122] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits – A design perspective*. 2nd. Prentice Hall, 2004.

[123] Rodric Rabbah, Ian Bratt, Krste Asanovic, and Anant Agarwal. “Versatility and VersaBench: A New Metric and a Benchmark Suite for Flexible Architectures”. In: (Dec. 2005).

[124] Jonathan Ragan-Kelley. *Decoupling Algorithms from the Organization of Computation for High Performance Image Processing*. Cambridge, MA, June 2014. URL: http://groups.csail.mit.edu/commit/papers/2014/jrkthesis.pdf.

[125] Praveen Raghavan, Satyakiran Munaga, Estela Ramos, Andy Lambrechts, Murali Jayapala, Francky Catthoor, and Diederik Verkest. “A Customized Cross-Bar for Data-Shuffling in Domain-Specific SIMD Processors”. In: *Proceedings of Architecture of Computing Systems (ARCS)*. Ed. by Paul Lukowicz, Lothar Thiele, and Gerhard Tröster. Vol. 4415. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, pp. 57–68. ISBN: 978-3-540-71267-1. URL: http://dx.doi.org/10.1007/978-3-540-71270-1_5.

[126] Ahmed Osman El-Rayis. *Reconfigurable architectures for the next generation of mobile device telecommunications systems*. PhD thesis. Nov. 2014.

[127] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. *Survey of Machine Learning Accelerators*. 2020. arXiv: 2009.00993 [cs.DC].

[128] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

[129] David E. Rumelhart and James L. McClelland. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. 1987, pp. 318–362.

[130] Mike Santarini. "Xilinx Ships Industry's First 20-nm All Programmable Devices". In: *Xcell Journal* (2014), pp. 9–15.

[131] S. Satpathy, Zhiyoong Foo, B. Giridhar, R. Dreslinski, D. Sylvester, T. Mudge, and D. Blaauw. "A 1.07 Tbit/s 128x128 swizzle network for SIMD processors". In: *Proceedings of IEEE Symposium on VLSI Circuits (VLSIC)*. June 2010, pp. 81–82. DOI: 10.1109/VLSIC.2010.5560282.

[132] S. Seo, M. Woh, S. Mahlke, T. Mudge, S. Vijay, and C. Chakrabarti. "Customizing wide-SIMD architectures for H.264". In: *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS '09. International Symposium on*. 2009, pp. 172–179. DOI: 10.1109/ICSAMOS.2009.5289229.

[133] Dongrui She, Yifan He, and Henk Corporaal. "Energy efficient special instruction support in an embedded processor with compact isa". In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. 2012, pp. 131–140.

[134] Dongrui She, Yifan He, B. Mesman, and H. Corporaal. "Scheduling for register file energy minimization in explicit datapath architectures". In: *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Mar. 2012, pp. 388–393.

[135] Dongrui She, Yifan He, Luc Waeijen, and H. Corporaal. "OpenCL code generation for low energy wide SIMD architectures with explicit datapath". In: *Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2013, pp. 322–329. DOI: 10.1109/SAMOS.2013.6621141.

[136] Dongrui She, Yifan He, Luc Waeijen, and Henk Corporaal. "OpenCL code generation for low energy wide SIMD architectures with explicit datapath". In: *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2013, pp. 322–329. DOI: 10.1109/SAMOS.2013.6621141.

[137] Savvas Sioutas, Sander Stuijk, Twan Basten, Henk Corporaal, and Lou Somers. "Schedule Synthesis for Halide Pipelines on GPUs". In: *ACM Trans. Archit. Code Optim.* 17.3 (Aug. 2020). ISSN: 1544-3566. DOI: 10.1145/3406117. URL: https://doi.org/10.1145/3406117.

[138] Savvas Sioutas, Sander Stuijk, Luc Waeijen, Twan Basten, Henk Corporaal, and Lou Somers. "Schedule Synthesis for Halide Pipelines through Reuse Analysis". In: *ACM Trans. Archit. Code Optim.* 16.2 (Apr. 2019). ISSN: 1544-3566. DOI: 10.1145/3310248. URL: https://doi.org/10.1145/3310248.

[139] M. Själander, H. Eriksson, and P. Larsson-Edefors. "An Efficient Twin-Precision Multiplier". In: *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.* Oct. 2004, pp. 30–33. DOI: 10.1109/ICCD.2004.1347894.

[140] Magnus Själander and Per Larsson-Edefors. "High-Speed and Low-Power Multipliers Using the Baugh-Wooley Algorithm and HPM Reduction Tree". In: *2008 15th IEEE International Conference on Electronics, Circuits and Systems*. Aug. 2008, pp. 33–36. DOI: 10.1109/ICECS.2008.4674784.

[141] Irvin Sobel. "Neighborhood coding of binary images for fast contour following and general binary array processing". In: *Computer Graphics and Image Processing* 8.1 (Aug. 1978), pp. 127–135. ISSN: 0146-664X. DOI: https://doi.org/10.1016/S0146-664X(78)80020-3.

[142] Standard Performance Evaluation Corporation. *SPEC CINT95 Benchmarks*. http://www.spec.org/cpu95/CINT95/index.html.

[143] Paul D. Stigall and Ömür Tasar. "A measure of computer flexibility". In: *Computers & Electrical Engineering* 2.2 (1975). ISSN: 0045-7906.

[144] Nai-Sheng Syu, Yu-Sheng Chen, and Yung-Yu Chuang. *Learning Deep Convolutional Networks for Demosaicing*. 2018. arXiv: 1802.03769.

[145] V. Sze, Y. -H. Chen, T. -J. Yang, and J. S. Emer. *Efficient Processing of Deep Neural Networks*. 2020.

[146] Tampere University of Technology. *TTA-based Codesign Environment (TCE)*. http://tce.cs.tut.fi/.

[147] GCC team. *GNU Compiler Collection*. Nov. 2021. URL: https://gcc.gnu.org/.

[148] Texas Instruments. "TMS320C6745, TMS320C6747 Fixed- and Floating-Point Digital Signal Processor". In: (2014).

[149] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norm Jouppi. *cacti 5.3, rev 174*. Available at http://quid.hpl.hp.com:9081/cacti/. Mar. 2014. URL: http://quid.hpl.hp.com:9081/cacti/.

[150] E. Tomusk, C. Dubach, and M. O'Boyle. "Measuring flexibility in single-ISA heterogeneous processors". In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. Aug. 2014, pp. 495–496.

[151] A. M. Turing. "Computers & Thought". In: ed. by Edward A. Feigenbaum and Julian Feldman. Cambridge, MA, USA: MIT Press, 1995. Chap. Computing Machinery and Intelligence, pp. 11–35. ISBN: 0-262-56092-5. URL: http://dl.acm.org/citation.cfm?id=216408.216410.

[152] Michel Van Lier, Luc Waeijen, and Henk Corporaal. "Bitwise Neural Network Acceleration: Opportunities and Challenges". In: *2019 8th Mediterranean Conference on Embedded Computing (MECO)*. June 2019, pp. 1–5. DOI: 10.1109/MECO.2019.8760178.

[153] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[154] Luc Waeijen. `ConvFuser`. https://gitlab.com/lwaeijen/convfusion.

[155] Luc Waeijen, Hailong Jiao, Henk Corporaal, and Yifan He. "Datawidth-Aware Energy-Efficient Multipliers: A Case for Going Sign Magnitude". In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. Aug. 2018, pp. 54–61. DOI: 10.1109/DSD.2018.00024.

[156] Luc Waeijen, Hailong Jiao, Henk Corporaal, and Yifan He. *Multiplier operand features - energy analysis data and tools*. https://git.ics.ele.tue.nl/luc/multiplier-predict.

[157] Luc Waeijen, Dongrui She, Henk Corporaal, and Yifan He. "A Low-Energy Wide SIMD Architecture with Explicit Datapath". In: *Journal of Signal Processing Systems* 80.1 (July 1, 2015), pp. 65–86. ISSN: 1939-8115. DOI: 10.1007/s11265-014-0950-8. URL: https://doi.org/10.1007/s11265-014-0950-8.

[158] Luc Waeijen, Dongrui She, Henk Corporaal, and Yifan He. "Reduction operator for wide-SIMDs reconsidered". In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2014, pp. 1–6. DOI: 10.1145/2593069.2593198.

[159] Luc Waeijen, Dongrui She, Henk Corporaal, and Yifan He. "SIMD made explicit". In: *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2013, pp. 330–337. DOI: 10.1109/SAMOS.2013.6621142.

[160] Luc Waeijen, Savvas Sioutas, Yifan He, Maurice Peemen, and Henk Corporaal. "Automatic Memory-Efficient Scheduling of CNNs". In: Springer

International Publishing, Aug. 2019, pp. 387–400. ISBN: 978-3-030-27561-7. DOI: 10.1007/978-3-030-27562-4_28.

[161] Luc Waeijen, Savvas Sioutas, Yifan He, Maurice Peemen, and Corporaal Henk. "Automatic Memory-Efficient Scheduling of CNNs". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation. SAMOS 2019. Lecture Notes in Computer Science*. Vol. 11733. 2019. DOI: https://doi.org/10.1007/978-3-030-27562-4_28.

[162] Luc Waeijen, Savvas Sioutas, Maurice Peemen, Menno Lindwer, and Corporaal Henk. "ConvFusion: A Model for Layer Fusion in Convolutional Neural Networks". In: *IEEE Access* 9 (2021), pp. 168245–168267. DOI: 10.1109/ACCESS.2021.3134930.

[163] J. van de Waerdt and et al. "The TM3270 media-processor". In: *Proceedings of the 38th International Symposium on Microarchitecture (MICRO)*. 2005, pp. 331–342.

[164] Gregory K. Wallace. "The JPEG Still Picture Compression Standard". In: *Commun. ACM* 34.4 (Apr. 1991), pp. 30–44. ISSN: 0001-0782.

[165] Nicolas Weber, Florian Schmidt, Mathias Niepert, and Felipe Huici. *BrainSlug: Transparent Acceleration of Deep Learning Through Depth-First Parallelism*. 2018. arXiv: 1804.08378 [cs.DC].

[166] Paul Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science. Thesis (Ph. D.). Appl. Math. Harvard University*. Jan. 1974.

[167] Mark Wijtvliet, Jos Huisken, Luc Waeijen, and Henk Corporaal. "Blocks: Redesigning Coarse Grained Reconfigurable Architectures for Energy Efficiency". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2019, pp. 17–23. DOI: 10.1109/FPL.2019.00013.

[168] Mark Wijtvliet, Luc Waeijen, Michaël Adriaansen, and Henk Corporaal. "Reaching intrinsic compute efficiency requires adaptable micro-architectures". English. In: 9th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG 2016), January 18, 2016, Prague, Czech Republic, MULTIPROG-2016 ; Conference date: 18-01-2016 Through 18-01-2016. Jan. 18, 2016, pp. 1–7. URL: http://research.ac.upc.edu/multiprog/.

[169] Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification". In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. July 2016, pp. 235–244. DOI: 10.1109/SAMOS.2016.7818353.

[170] Rand R. Wilcox and H. J. Keselman. “Modern robust data analysis methods: measures of central tendency.” In: *Psychological methods* 8 3 (2003).

[171] Markus Willems. *Application-Specific Processors for High Throughput, Low Latency, and Flexible 5G Communication SoCs*. Synopsis, 2019. URL: https://www.synopsys.com/designware-ip/technical-bulletin/5g-asips-communication-socs.html (visited on 03/27/2021).

[172] M. Woh, Sangwon Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. “AnySP: Anytime Anywhere Anyway Signal Processing”. In: *IEEE Micro* 30.1 (Jan. 2010), pp. 81–91. ISSN: 0272-1732. DOI: 10.1109/MM.2010.8.

[173] M. Woh et al. “From SODA to scotch: The evolution of a wireless baseband processor”. In: *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*. Nov. 2008, pp. 152–163. DOI: 10.1109/MICRO.2008.4771787.

[174] Yannan N. Wu, Joel S. Emer, and Vivienne Sze. “Accelergy: An Architecture Level Energy Estimation Methodology for Accelerator Designs”. In: *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.

[175] Yannan N. Wu, Joel S. Emer, and Vivienne Sze. *Accelergy: An Architecture Level Energy Estimation Methodology for Accelerator Designs - Slides*. http://accelergy.mit.edu/slides.pdf.

[176] Yannan N. Wu, Amin A. Ghasemazar, and Po-An Tsai. *Accelergy-Aladdin-Plug-in*. https://github.com/Accelergy-Project/accelergy-aladdin-plug-in.

[177] Wm. A. Wulf and Sally A. McKee. “Hitting the Memory Wall: Implications of the Obvious”. In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588. URL: https://doi.org/10.1145/216585.216588.

[178] Xilinx. “Vivado Design Suite User Guide: High-Level Synthesis”. In: (Apr. 2017).

[179] Jun Yan and Wei Zhang. “Virtual registers: Reducing register pressure without enlarging the register file”. In: *Proceedings of High Performance Embedded Architectures and Compilers (HiPEAC)*. 2007, pp. 57–70.

[180] Menghui Zheng and A. Albicki. “Low power and high speed multiplication design through mixed number representations”. In: *Computer Design: VLSI in Computers and Processors, 1995. ICCD ’95. Proceedings., 1995 IEEE International Conference on*. Oct. 1995, pp. 566–570. DOI: 10.1109/ICCD.1995.528924.

[181] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. *FusionStitching: Boosting Memory Intensive Computations for Deep Learning Workloads*. 2020. arXiv: 2009.10924 [cs.DC].

[182] Διόδωρος Σικελιώτης. *The Library of History of Diodorus Siculus, Fragments of Book XXVI*. URL: https://penelope.uchicago.edu/Thayer/E/Roman/Texts/Diodorus_Siculus/26*.html.

Publications

First Author

1. Luc Waeijen, Dongrui She, Henk Corporaal, and Yifan He. "SIMD made explicit". In: *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2013, pp. 330–337. DOI: 10.1109/SAMOS.2013.6621142

2. Luc Waeijen, Dongrui She, Henk Corporaal, and Yifan He. "A Low-Energy Wide SIMD Architecture with Explicit Datapath". In: *Journal of Signal Processing Systems* 80.1 (July 1, 2015), pp. 65–86. ISSN: 1939-8115. DOI: 10.1007/s11265-014-0950-8. URL: https://doi.org/10.1007/s11265-014-0950-8

3. Luc Waeijen, Dongrui She, Henk Corporaal, and Yifan He. "Reduction operator for wide-SIMDs reconsidered". In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2014, pp. 1–6. DOI: 10.1145/2593069.2593198

4. Luc Waeijen, Savvas Sioutas, Yifan He, Maurice Peemen, and Corporaal Henk. "Automatic Memory-Efficient Scheduling of CNNs". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation. SAMOS 2019. Lecture Notes in Computer Science.* Vol. 11733. 2019. DOI: https://doi.org/10.1007/978-3-030-27562-4_28

5. Luc Waeijen, Savvas Sioutas, Maurice Peemen, Menno Lindwer, and Corporaal Henk. "ConvFusion: A Model for Layer Fusion in Convolutional Neural Networks". In: *IEEE Access* 9 (2021), pp. 168245–168267. DOI: 10.1109/ACCESS.2021.3134930

6. Shihua Huang, Luc Waeijen, and Henk Corporaal. "How Flexible is Your Computing System?" In: *ACM Trans. Embed. Comput. Syst.* (Mar. 2022). ISSN: 1539-9087. DOI: 10.1145/3524861. URL: https://doi.org/10.1145/3524861

7. Luc Waeijen, Hailong Jiao, Henk Corporaal, and Yifan He. "Datawidth-Aware Energy-Efficient Multipliers: A Case for Going Sign Magnitude". In:

2018 21st Euromicro Conference on Digital System Design (DSD). Aug. 2018, pp. 54–61. DOI: 10.1109/DSD.2018.00024

Co-authored

1. Dongrui She, Yifan He, Luc Waeijen, and Henk Corporaal. "OpenCL code generation for low energy wide SIMD architectures with explicit datapath". In: *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2013, pp. 322–329. DOI: 10.1109/SAMOS.2013.6621141

2. Yifan He, Maurice Peemen, Luc Waeijen, Erkan Diken, Mattia Fiumara, Gerard Rauwerda, Henk Corporaal, and Tong Geng. "A configurable SIMD architecture with explicit datapath for intelligent learning". In: *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. July 2016, pp. 156–163. DOI: 10.1109/SAMOS.2016.7818343

3. Tong Geng, Luc Waeijen, Maurice Peemen, Henk Corporaal, and Yifan He. "MacSim: A MAC-Enabled High-Performance Low-Power SIMD Architecture". In: *2016 Euromicro Conference on Digital System Design (DSD)*. Aug. 2016, pp. 160–167. DOI: 10.1109/DSD.2016.27

4. Mark Wijtvliet, Jos Huisken, Luc Waeijen, and Henk Corporaal. "Blocks: Redesigning Coarse Grained Reconfigurable Architectures for Energy Efficiency". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2019, pp. 17–23. DOI: 10.1109/FPL.2019.00013

5. Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification". In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. July 2016, pp. 235–244. DOI: 10.1109/SAMOS.2016.7818353

6. Mark Wijtvliet, Luc Waeijen, Michaël Adriaansen, and Henk Corporaal. "Reaching intrinsic compute efficiency requires adaptable micro-architectures". English. In: 9th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG 2016), January 18, 2016, Prague, Czech Republic, MULTIPROG-2016 ; Confer-

ence date: 18-01-2016 Through 18-01-2016. Jan. 18, 2016, pp. 1–7. URL: http://research.ac.upc.edu/multiprog/

7. Michaël Adriaansen, Mark Wijtvliet, Roel Jordans, Luc Waeijen, and Henk Corporaal. "Code Generation for Reconfigurable Explicit Datapath Architectures with LLVM". in: *2016 Euromicro Conference on Digital System Design (DSD)*. Aug. 2016, pp. 30–37. DOI: 10.1109/DSD.2016.88

8. Stef Louwers, Luc Waeijen, Mark Wijtvliet, Ruud Koolen, and Henk Corporaal. "Multi-granular Arithmetic in a Coarse-Grain Reconfigurable Architecture". In: *2016 Euromicro Conference on Digital System Design (DSD)*. Aug. 2016, pp. 599–606. DOI: 10.1109/DSD.2016.98

9. Michel Van Lier, Luc Waeijen, and Henk Corporaal. "Bitwise Neural Network Acceleration: Opportunities and Challenges". In: *2019 8th Mediterranean Conference on Embedded Computing (MECO)*. June 2019, pp. 1–5. DOI: 10.1109/MECO.2019.8760178

10. Savvas Sioutas, Sander Stuijk, Luc Waeijen, Twan Basten, Henk Corporaal, and Lou Somers. "Schedule Synthesis for Halide Pipelines through Reuse Analysis". In: *ACM Trans. Archit. Code Optim.* 16.2 (Apr. 2019). ISSN: 1544-3566. DOI: 10.1145/3310248. URL: https://doi.org/10.1145/3310248

11. Mike Jongen, Luc Waeijen, Roel Jordans, Lech Jozwiak, and Henk Corporaal. "Optimization through recomputation in the polyhedral model". English. In: *Eighth International Workshop on Polyhedral Compilation Techniques*. 8th International Workshop on Polyhedral Compilation Techniques (IMPACT 2018), January 23, 2018, Manchester, UK, IMPACT ; Conference date: 23-01-2018 Through 23-01-2018. Jan. 22, 2018. URL: http://impact.gforge.inria.fr/impact2018

12. Jarno Brils, Luc Waeijen, and Arash Pourtaherian. "How to Exploit Sparsity in RNNs on Event-Driven Architectures". In: *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '21. Eindhoven, Netherlands: Association for Computing Machinery, 2021, pp. 17–22. ISBN: 9781450391665. DOI: 10.1145/3493229.3493302. URL: https://doi.org/10.1145/3493229.3493302

13. Lennart Bamberg, Arash Pourtaherian, Luc Waeijen, Anupam Chahar, and Orlando Moreira. *Synapse Compression for Event-Based Convolutional Neural Network Accelerators*. 2021. arXiv: 2112.07019 [cs.AR]

Acknowledgments

Nothing worth achieving can be achieved alone. The thesis you have been reading would not have come to be if not for the guidance, time, effort, and help, of many. In this section I would like to express my gratitude to those who supported me both in technical, and non-technical ways.

First of all I would like to thank my promotor, Henk Corporaal, from who I learned an incredible amount during these past years. Henk genuinely loves what he does, which supplies him with a seemingly unlimited source of new and interesting ideas that have kept me inspired (and well occupied) throughout my PhD. His ability to ask critical questions about work only just presented to him, has never failed to amaze me, and is what has enabled me to develop as a researcher.

Another person I can not thank enough is my copromotor, Yifan He, who I first got to know in his role as supervisor during my master thesis work. He enabled me to write my first paper which heavily built upon his and Dongrui She's work, and he continued to mentor me after I graduated and started my PhD. Those early years were incredibly productive, and Yifan played a key role in that success. I am extremely grateful for his guidance and kindness.

Additionally I would like to thank Prof. Marian Verhelst, Prof. Rob van Nieuwpoort, Dr. Sander Stuijk, Dr. Maurice Peemen, and Dr. Savvas Sioutas, for being part of my doctoral committee, and for their time and effort spent reviewing the manuscript.

I would also like to thank my current employer, GrAI Matter Labs, for providing me with the means required to finish my PhD next to my job, and their continued support even through busy (but exciting!) times at the company. A special thanks is in order to Menno Lindwer, who sacrificed a substantial amount of his own time trying to get this stubborn perfectionist to finish his thesis. Without him I would still be rewriting my tools and running additional superfluous experiments.

I have had the questionable honour to see several generations of PhDs come and go during my time at the university. Despite the obvious implication that I have overstayed my welcome, I do consider it a privilege that I had the opportunity to get to know so many smart and kind people. Alessandro, Ali, Amir & Hadi (these

two can not be separated), Andreia, Ang, Andrew, Barry, Berk, Bram, Cedric, Cumhur, Dongrui, Firew, Gabriela, Gagandeep, Gert-Jan, Hadi 2, Hamideh, Ilde, Joost, João, Juan, Kanishkan, Kamlesh, Mahsa, Martijn, Martin, Maurice, Mladen, Mohammad, Paul, Rasool, Reinier, Robinson, Roel, Ruben, Sajid, Savvas, Shakith, Shreya, Shima, Shubendhu, Sven, Tong, Umar, Yifan, and Zhenyu, thank all of you for making my time at the ES-group unforgettable. The same holds for the staff of the ES-group, who had put up with me even longer than anyone in the list above: Henk, Sander, Twan, Kees, Dip, Lech, Marc, Hailong, and Marja. Finally a special thanks to Martijn for maintaining the group's servers and trusting me with more super powers than I should have had access to, and Juan for motivating me to trade my office chair for the swimming pool or bike a bit more often.

During my PhD, I also had the distinct honour to work with, and corrupt the minds of, many clever students who I would like to retroactively apologize to: Adrian Guillot, Boyan Liang, Chinmay Nemade, Esther Dommisse, Guus Leijsten, Jarno Brils, Jiachi Zou, Kanishkan Vadivel, Matthias Schneider, Michaël Adriaansen, Michel van Lier, Mike Jongen, Shihua Huang, Stef Louwers, Steven Hunsche, Victor Fornade, and Zhenyuan Liu. A special thanks to Shihua Huang, who spent many evening hours even after her graduation to help shape the work presented in chapter 6.

I believe it to be utterly impossible to complete a PhD thesis and somewhat maintain one's sanity without regularly sharing a drink (or two) with good company. Francesco and João, thanks for dragging me out of the office and towards "kleine berg" in those early years, and teaching me some proper southern European etiquette. The start of a Friday evening tradition that was carefully continued with Savvas, Ilde, Alessandro, Sayandip, Kamlesh, Shima, Paul and Barry. Our evenings at Loods, and the many home cooked Italian meals by Chef Ale, have been as enjoyable as they have been necessary. In the context of sharing drinks, also a word of gratitude to the volunteers of the Wahalla where many hours and beers were spent recovering from the work week, in particular with the infamous brothers Cox: Robbert, and my fellow Rustyman Marco.

I can not mention Marco and forget about my lunch-foster-group from SPS who so kindly adopted me during many lunches: Thijs, Ivan, and Anton. Thanks for all the laughs we shared, but for the sake of humanity and cats; let's never start a business together.

Being an expat in Eindhoven, I have also been blessed with the support of my friends and family back in my home country Limburg. My loving parents,

who somehow always found the strength to support me, no matter how self-detrimental my choices seemed to be. My brother Joop, his wife Marjolein, and 'the rest of the gang': Ruben, Stefanie, Manon, Lotte, and Ben, who took me in many times when I had found my way back home from Eindhoven. Geert, my oldest friend with whom I have explored both technology & life for as long as I can remember, and hope to continue doing that with for a very long time to come. And of course, Mieke and Jill, for supplying me with my regular dose of sarcasm and witty humour when I needed it most.

Load-balancing is a difficult task, and there are some people that I disproportionally burdened with my problems and presence. Marco, our coffee break conversations were never dull, and without your moral support I would have been utterly lost on more than one occasion. Savvas, my Greek brother (he is the smarter one), who is always there to serve as an amplifier for my pessimism (or is it realism?), and remind me that nobody cares. And last but not least, Anouk, who I arguably damaged the most by convincing her to start a relationship with me during the most challenging years of my PhD journey. You have seen me at my worst, when I was demotivated, agitated, and too tired to do anything fun. It is beyond me how you managed to not only put up with me, but actively help me find the required time and peace of mind to complete my PhD. You are next, just look at what I did, and invert absolutely everything!

Finally, I do not want to pass up on this opportunity to thank each and everyone who persistently asked me how my thesis was coming along. I profoundly hated every time you asked, although that was kind of the point I suppose, so thank you for that too.

καλά όλα αυτά!

Luc Waeijen
Eindhoven, September 2022

About the Author

Luc Johannes Wilhelmus Waeijen was born in Roermond, the Netherlands, on December 17, 1988. He received the BSc degree in Electrical Engineering from Eindhoven University of Technology in 2012.

In 2013 he obtained his MSc degree in Embedded Systems from Eindhoven University of Technology. The focus of his Master thesis was research into the wide SIMD featured in the first chapters of this thesis.

After graduating, Luc joined the Electronic Systems (ES) group at Eindhoven University of Technology to pursue a PhD in computer architecture, the results of which are presented in this thesis.

In 2019 he left the ES group and joined GrAI Matter Labs in Eindhoven. Currently he is with the Architecture Group at GrAI Matter Labs, driving the design and specification of data-driven AI processors.