Master's thesis

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

Lasse Agentoft Eggen

Towards Efficiently Utilizing Coarse-Grained Reconfigurable Accelerators

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

NTNU
Norwegian University of
Science and Technology

Lasse Agentoft Eggen

Towards Efficiently Utilizing Coarse-Grained Reconfigurable Accelerators

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

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

NTNU
Norwegian University of
Science and Technology

Problem Description

The end of Dennard scaling and the imminent end of Moore's law is causing disruptive changes to the way computers are designed. An attractive option is to create specialized hardware units – called accelerators – that are able to execute performance-critical code regions (much) more efficiently than a general-purpose processor. Unfortunately, designing accelerators is costly which limits their applicability to high-volume domains such as graphics or machine-learning. Another option is to add tightly coupled reconfigurable fabrics to general-purpose processors and combine this with efficient approaches for generating application-specific accelerators.

In this thesis, the student should work towards enabling such systems by investigating how programs represented as dataflow graphs can be efficiently mapped onto a Coarse-Grained Reconfigurable Array (CGRA). Mapping computations onto CGRAs is challenging because they expose limited reconfigurability to retain efficiency. Further, prior work has shown that mapping dataflow graphs to small CGRAs can be particularly challenging because naive mapping can cause extensive serialization. The student should first try to address this issue with static approaches. If time permits, software-hardware cooperative approaches can be explored.

Abstract

The end of Dennard scaling and the imminent end of Moore's law is causing disruptive changes to the way computers are designed. There is agreement in the computer-architecture community that we probably need specialized hardware to further improve performance and increase power and energy efficiency. The industry standard is domain-specific accelerators that accelerate a common set of applications, at the cost of generality. An alternative is reconfigurable architectures which can accelerate multiple domains and thereby achieve better utilization. In this thesis, we focus on Coarse-Grained Reconfigurable Architectures (CGRAs), as they have higher theoretical performance than fine-grain alternatives, such as Field-Programmable Gate Arrays (FPGAs).

We studied CGRA accelerators in the context of state-of-the-art Stream-Dataflow Architecture (SDA), and the Reconfigurable Vector Lanes (REVEL) accelerator. First, we model a set of different CGRA sizes to explore performance-scaling opportunities. Then, by varying the ratio of static-to-dynamic scheduling we assess the performance impact of a dynamic-dataflow region and to what extent the mapping algorithm is able to exploit it. Finally, we explore how performance can be improved by factoring in an active developer.

We find that attaining high performance requires that the program formulation, the mapping and scheduling algorithms, and the Coarse-Grained Reconfigurable Array (CGRA) accelerator architecture all align favorably. Due to the high complexity in the mapping and scheduling problem, we are not able to gain efficiency when we try to express more parallelism than the original REVEL workloads. We have shown empirically that increasing the CGRA size alone does not contribute to execution scalability, neither does introducing dynamic dataflow in isolation. Software-hardware cooperation is hence key to maximize the performance of CGRA accelerators. Unfortunately, we were not able to qualitatively evaluate such approaches with the constraints of a master thesis due to limitations with our chosen compiler and simulator framework (even if it is the current state-of-the-art).

Contents

Figures

Tables

Code Listings

Acronyms

ASIC Application-Specific Integrated Circuit. 6, 10–12, 19

CGRA Coarse-Grained Reconfigurable Array. v, xi, 2–5, 8, 11–13, 15, 17–22, 24–31, 34, 35, 37–39, 41

CPU Central Processing Unit. 2, 3, 13, 25

DFG Dataflow Graph. 3, 4, 8, 13, 15, 17, 20–26, 31, 35, 36, 38, 42

DNN Deep Neural Network. 10, 11, 15

DSA Domain-Specific Accelerator. 2, 5, 6, 9–12, 19, 20

FPGA Field-Programmable Gate Array. 2, 11–13

FU Functional Unit. 9–13, 15, 16, 20–23, 26, 35

GPP General Purpose Processor. 5, 6, 9, 10, 35, 36

GPU Graphics Processing Unit. 2, 3, 9

HLS High-Level Synthesis. 11

PE Processing Element. xi, 12, 13, 15–18, 20–23, 25, 27–29, 31, 35, 36, 38, 41

RTL Register-Transfer Level. 11, 12

SDA Stream-Dataflow Architecture. 19, 20

TPU Tensor Processing Unit. 10, 12

Chapter 1

Introduction

1.1 Motivation

From the early beginnings of computing, improvements in technology has allowed almost free scaling of performance with similar architecture designs, as illustrated in Figure 1.1. Then conventional computer architecture was hit hard by laws of physics that dictate attainable performance with factors such as voltage, feature size and frequency. Until mid 1990s power budgets in chips was not a major design constraint for microarchitectures, when it started to be widely recognized that power budgets in computer circuits had to be addressed at a higher level of abstraction to allow for further improvements [2].

This realization is commonly referred to as the power wall, and is a direct consequence of the end of Dennard voltage scaling [3], which boils down to the fact that technology-node improvements directly influenced the ability to increase frequency and being able to offset the power dissipation by decreasing the voltage supply. This automatic performance improvement drove microarchitecture designs, which was essentially just distributing more of the same logic on the same die area, and increasing clock frequency.

Today we are no longer able to decrease the threshold voltage as this exponentially increases leakage currents, making it a significant contributor to power dissipation [4]. Without the possibility to offset increasing clock frequencies with lower voltage, CMOS-based chips are subject to overheating. The result of this is dark silicon. Dark silicon is circuitry that can not switch at full frequency due to power limitations and/or high thermal density. When this is the case, only parts of the processor will be active at any given time.

Recent studies has shown that multicore scaling will also suffer from increased amounts of dark silicon, just as the case was for single-core scaling [5]. Increasing the amount of parallelism has become the most important factor in increasing performance and energy efficiency for general programs in recent years. Instead of building a single core with more transistors and higher clock rate to increase single-core performance, the industry's answer has been to partition the area into multiple processor cores. These cores run at lower clock rates than e.g. Intel Pen-

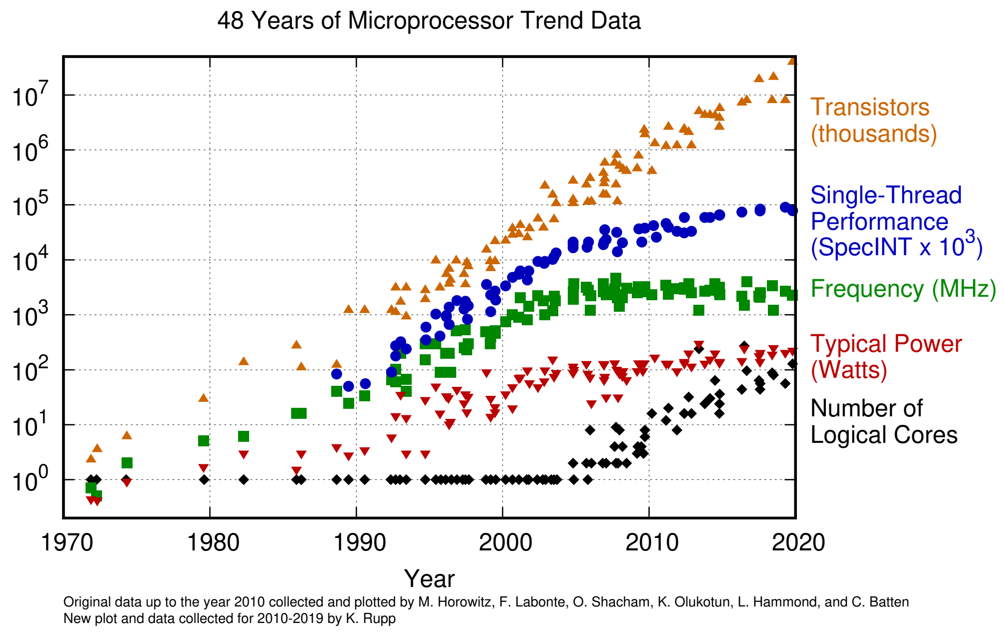


Figure 1.1: 48 Years of Microprocessor Trend Data [1].

tium 4, but since there are more cores, more work can be done at the same time.

1.2 Reconfigurable Accelerator Overview

The most promising approach to the dark silicon problem is hardware specialization. Specializing hardware has given high efficiency and high performance for a multitude of applications, and the more specialized the hardware is, the more efficient it becomes. Domain-Specific Accelerators (DSAs) that aim to accelerate a specific domain of problems deliver both of these, at the cost of low usability of the hardware outside its domain. They are also expensive to design and produce, making sense mostly when they can be mass produced.

An alternative is reconfigurable accelerators. The classic Field-Programmable Gate Array (FPGA) is very flexible and can be programmed to host mostly any class of program. The flexibility comes from reprogrammability at the bit-level, which turns out to be its least attractive attribute. According to Taylor [6], CGRAs are more promising than FPGAs to realize the dim horseman, due to expectations of high energy efficiency. The layout of the architecture is broadly speaking optimized for word-level operations that require a relatively short and small amount of interconnects, and contains highly optimized computational hardware. Data travels naturally through this network of computation, while minimizing multiplexing, which helps gain high performance per watt and area.

A CGRA will be more efficient and able to exploit more parallelism from applications than Central Processing Units (CPUs) and Graphics Processing Units

(GPUs). A CPU will not be able to exploit parallelism well, it turns out that concurrent programming is hard [7], and efficiency is sacrificed for generality [8]. A GPU, on the other hand, will be able to match CGRA performance for some applications, but is far less power and energy efficient [9].

1.3 Assignment Interpretation

My interpretation of the assignment text in Problem Description is to pursue a specific programming model, specifically Dataflow Graph (DFG)-based program kernel representations, and their potential for mapping to a CGRA. More specifically, I interpret the assignment text to contain the following tasks:

T1 How effective are purely static (systolic array) approaches?
T2 How can dynamic dataflow overcome this limitation?
T3 How does things change if the developer (or compiler) takes an active role (i.e., a software-hardware cooperative approach)?

These tasks are not sufficiently narrowly defined to be fully addressed within a master thesis. Hence, I further narrow the scope to focus on what makes a specific representation able to efficiently map to a varying number of substrate configurations.

There is primarily two execution models that I devote my attention, fully static and fully dynamic scheduling. These models are on both ends of the execution-model spectrum and I have previous knowledge from the fall-semester project handling a hybrid architecture, namely the REVEL architecture [10]. The project found that the temporal region extensively serialized DFGs that did not fit the dedicated region. In contrast, this work will focus on the effectiveness of purely dedicated configurations, then on what dynamic dataflow contributes, and what effects a software-hardware cooperative approach can enable in terms of performance scaling.

1.4 Contributions

In this thesis, we respond to the three key tasks provided in the assignment text. Hence, the key contributions of this thesis are:

C1 We found that a pure static approach to scheduling is not scalable. More specifically, we find that attaining high performance requires that the program formulation, the mapping and scheduling algorithms, and the CGRA accelerator architecture all align favorably. This solution responds to T1.
C2 Dynamic (hardware) scheduling simplifies mapping applications onto the CGRA accelerator. While this enables mapping applications that would not map to a static CGRA accelerator, it comes at the cost of increased hardware overhead. This solution responds to T2.

C3 Software-hardware cooperation enables scaling of computation on statically scheduled regions and hence has the potential for overcoming the aforementioned limitations. However, this requires substantial manual effort, and we find that the current state-of-the-art tool, notably the PolyArch framework [11–13], struggle with efficiently mapping DFGs to larger CGRAs. This solution responds to T3.

1.5 Outline

In Chapter 2, we describe traits of typically acceleratable programs and introduce a series of existing programmable, and reconfigurable accelerator approaches. Then, in Chapter 3 we describe systolic array, static dataflow, and dynamic dataflow. In Chapter 4, we introduce the REVEL architecture we will simulate and collect data from, and show how a DFG can map to a CGRA, with simple improvement strategies. Then, in Chapter 5 we show how we have configured the architecture and the simulator setup. We show results in Chapter 6, analyzing program performance against CGRA size, and temporal-region size. Then, in Chapter 7 we list the key lessons we learned, namely that performance scaling is not automatic. Finally, we conclude in Chapter 8.

Chapter 2

Background

From fixed-function processing elements in GPPs to per-cycle reconfigurable processing elements in CGRAs, there are a wide variety in methods for transforming data as efficiently as possible in terms of chip area, time, power and energy. In Figure 2.1 we illustrate the programming generality of a set of processor architectures. The more general an architecture is, the less specialized it is, conversely the opposite is also true.

GPPs interpret a sequence of instructions and are software-driven. This makes them easy to program and highly flexible and general, at the cost of efficiency. Typically, the more flexible and easy to program a processor is, the less efficient it is in terms of area, power and energy. On the other extreme, we have DSAs, processors that trade off flexibility and programmability to achieve high efficiency.

2.1 Accelerating Applications

Amdahl's law is useful to understand what we want to focus our attention on when analyzing programs as candidates for acceleration [14]. In this work we will explore parallelism in a program that can be exploited to obtain good speedup of the program. We will also follow the corollary that the fraction of a program we want to improve has to be relatively large, if not we will not obtain significant speedup.

As an example, imagine a program which execution time is dominated by sequential code and we can only identify 5% of the code's runtime to be acceleratable. If all else remains equal, we can only achieve a speedup of $1.05x$, and with marginal gains every $2x$ speedup of the fractional part we are able to improve on.

2.1.1 Characteristics of Accelerator-Friendly Applications

Nowatzki et al. [15] [11] argue that a set of characteristics found in acceleratable programs are those that can be found in applications that target DSAs. We too find these interesting, as we can relate to the idea that DSAs have been co-designed with digital-design algorithms. The characteristics are as follows:

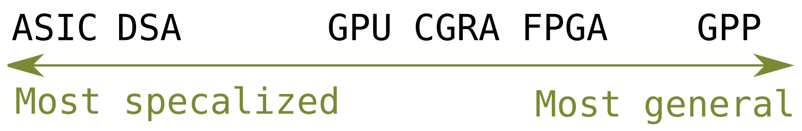


Figure 2.1: General and special architectures. The most specialized architectures, Application-Specific Integrated Circuit (ASIC), and DSA, are the least general. The least specialized architecture, General Purpose Processor (GPP), is the most general.

1. Inhibit significant thread-level or data-level parallelism.
2. High computational intensity with long phases.
3. Units of work are coarse grained, and simple control flow.
4. Straight-forward memory access and reuse patterns.

Exploitable parallelism is key to achieve higher performance. Without parallelism we can not expect to speedup an application much. Typical applications that have high levels of parallelism are those that have a high amount of independent computations, such as vector and matrix operations. These applications usually inhibit **high computational intensity with long phases**. High computational intensity refers to how many operations we perform per byte of data we transfer from memory. The more operations an algorithm perform per byte, the more efficient the accelerator can operate.

Further, **coarse-grained units of work** in this context refers to a contiguous chain of operations that easily can be pipelined. Typically inner loops, that also can be unrolled, ultimately vectorized in hardware. **Control flow** often means resolving branches or limiting the amount of parallelism we can extract from an application, which interferes with our goal of accelerating the application.

Memory access should be regular to allow streaming data from memory. Irregular memory access will likely reduce computational intensity as we might have to transfer more bytes than we will use. Reuse patterns are desirable to keep data either in the accelerator fabric or very close, such as in a scratchpad. Not all applications are designed with these characteristics in mind, so it is possible to discover or impose these traits in applications that at first sight does not appear to inhibit them. This is true for many DSAs [15].

What ultimately makes an accelerator worthwhile is that it exposes the ability to specialize hardware to gain efficiency while opening possibilities for speedup through parallelism. Most applications that lend themselves to acceleration in terms of speedup through parallelization eventually hit a memory wall, i.e. there is abundant computational resources, but the memory system is unable to deliver the data to the accelerator fast enough.

2.1.2 Example Kernels

We have listed three program kernels in Code listing 2.1, Code listing 2.2 and Code listing 2.3 that we will use throughout this thesis to show how we can ex-

Code listing 2.1: Vector addition

```
void vector_add(double* a, double* b, double* c) {
  #pragma ss config
  {
    #pragma ss stream
    #pragma ss dfg dedicated unroll(4)
    for (int i = 0; i < N; ++i) {
      c[i] = a[i] + b[i];
    }
  }
}
```

Code listing 2.2: Vector dot product

```
double dot_product(double* a, double* b) {
  double dot;
  #pragma ss config
  {
    double acc = 0.0;
    #pragma ss stream
    #pragma ss dfg dedicated unroll(4)
    for (int i = 0; i < N; ++i) {
      acc += a[i] * b[i];
    }
    dot = acc;
  }
  return dot
}
```

Code listing 2.3: Inductive dependency. Inner-loop's induction variable, j, depends on outer-loop's induction variable, i.

```
double inductive_dependency(double* a, double* b, double* c) {
  #pragma ss config
  {
    #pragma ss stream
    for (int i = 0; i < N; ++i) {
      #pragma ss temporal
      {
        double x = sqrt(a[i] + b[i]);
      }
      #pragma ss dedicated unroll(4)
      for (int j = i+1; j < N; ++j) {
        c[i] += a[j] * b[i] + x;
      }
    }
  }
}
```

tract parallelism and gain efficiency on different architectures. Vector addition is a trivial example when we can stream data to the architecture, while dot product requires some thought to compute efficiently.

In Code listing 2.1, vectors a and b are added and the result is stored in c. This program lends itself to trivial pipelining and unrolling. I.e. all N additions can, in theory, be unrolled and calculated in parallel (i.e., spatial parallelism), and all operations in the loop can be done in a pipelined manner (i.e., temporal parallelism). As there is only one operation per loop iteration, an addition, there is no need to pipeline anything, but for the sake of argument this pipeline has a length of one. The kernel has low computational intensity, one operation per element of data, and will most likely be memory bound.

Typically we need to annotate code to mark blocks as candidates for parallel execution, such as CUDA [16], OpenMP [17], and OpenCL [18]. Often annotation is done by inserting `#pragma` directives. We are focusing on CGRA programming and have used `#pragma` directives from REVEL [12] (see Chapter 4 for more detail). Lines 2, 4 and 5 are pragmas that are introduced by the REVEL compiler toolchain. Line 2 dictates that the following block is a candidate to create an accelerator that is ultimately offloaded to a reconfigurable substrate, line 4 initializes a memory stream for the following loop, and line 5 marks the same loop as a parallelization candidate.

The kernel in Code listing 2.2 calculates the dot product of two vectors. It is less straight forward to accelerate efficiently as it requires reduction of partial results if unrolled to extract parallelism, and some way to store the running state of the accumulator. In the `dot_product()` function an accumulator is used to store a running state of the program and is returned at the end. One might wonder why this program has two variables that ultimately holds the same dot product value, the reason is that the DFG generator and REVEL compiler toolchain allocates acc in the accelerator and uses accumulation logic to store the running sum and releases the final dot product from the CGRA to the function's dot variable. The kernel inhibits quite low computational intensity and will most likely be memory bound.

In Code listing 2.3 we show a sample kernel that has data dependencies and a dependency on an induction variable. In the outer loop we calculate the square root of the sum of two elements. This result is reused $N-j$ times in the inner loop, unlike the regular patterns we see in the other kernels. The inner loop's induction variable, j, is dependent on the outer loop's induction variable, i. When these patterns are present, the inner loop's amount of work varies with the outer loop's counter variable, resembling a triangular shape.

2.2 Programmable Accelerators

2.2.1 Graphics Processing Units

What started out as DSAs to accelerate graphics, GPUs, has in the last few decades proven effective at accelerating a host of problems, from machine learning [19] with examples such as ImageNet [20] to blockchain [21]. Since everyone had a GPU, or easily could acquire one, they quickly became popular to accelerate many applications. Acceleration of these problems has long been favorable, even though the energy efficiency of GPUs are poor.

A GPU excel at multi-threading, it is a high-throughput processor that can perform hundreds of independent computations every cycle. Requesting data from memory is a high-latency operation, and being able to compute hundreds of operations per cycle is irrelevant if no data is available. The GPU hides memory latency by rapid, essentially free, context switches. To reach maximum utilization of a GPU we want to saturate the memory bus and a very high amount of threads.

Simplifying a bit, the GPUs executes threads in warps. A warp is a collection of 32 threads that simultaneously execute the same instruction. This implies that GPUs suffer from control and memory divergence [22]. All threads execute the same program and when they reach a branch, all threads have different contexts and resolve the branch independently. When a number of the threads diverge from the others, a partition of the threads are masked out of execution. High divergence leads to low utilization, while still consuming the same amount of power.

Considering our example kernel, the vector addition in Code listing 2.1, we can quickly see that this is a good candidate to target a GPU. It independently calculates data from contiguous locations of memory and writes results to predictable locations. These independent calculations can all be distributed to a large collection of concurrent threads without dependencies. It is also free of control flow and there is no chance of control divergence. We expect that we can accelerate this kernel easily on a GPU.

2.2.2 Domain-Specific Accelerators

The introduction of DSAs is a reaction to the lack of progress in performance improvements in GPPs. The key idea is that more efficient computing is possible by relaxing the one-size-fits-all approach of GPPs. For DSAs this is done by restricting generality in the programming interface which enables less complex hardware resulting in increased efficiency. Performance gain is achieved through extensive parallelization.

We highlight five major techniques to increase efficiency and performance in DSAs based on Dally [23] and Nowatzki et al. [24]:

1. **Specialized data types and operations:** Increases performance by a factor of 10x to 100x by consolidating memory layout and execution units. This can by achieved by specializing Functional Units (FUs), to reduce power

and increase performance by reducing total work.

2. **Massive parallelism:** In conjunction with improving memory locality this effect may increase performance by 1000x.

3. **Optimized memory:** Specialized memory designed to be energy efficient when transmitting specific data structures and specific target operations in mind.

4. **Reduced overhead:** The overhead can be reduced both on a sequential basis or amortized over a batch of actions or operations.

5. **Algorithm-architecture co-design:** To reach maximum performance and energy efficiency, the algorithm most likely has to be rewritten to release it from the shackles of instruction-based computing in GPPs. If it is designed with the four previous techniques in mind, the accelerator will attain high performance.

Bleeding-edge iphones house 40+ accelerators, and is an example of how important specialization has become when designs are limited by power and energy [25]. Since domains and applications vary in organization, the number of DSA units needed to accelerate different problems will claim a lot of chip real estate. How to decide on a mix of DSAs can be relatively easy for simple embedded systems, but integrating the accelerators alongside a GPP might not be possible.

A full-custom DSA is the best option when considering area, performance, power and energy. They can be fine-tuned to a specific application. The major drawbacks are the costs of development, validation and time, in addition to the possibility of the units being made obsolete by new or improved algorithms. To produce ASICs there normally has to be a benefit in terms of economy. Mass producing ASIC chips is generally the only way to amortize the engineering costs, the target domain needs to be so large that the demand is in the order of million chips. Post-production they are masters of their task, but irregardless of the high-volume requirement, researchers have proposed DSAs for a number of domains.

In recent years, Google's Tensor Processing Unit (TPU) [26] is a state-of-the-art example of an industrial-level DSA. The processor optimizes Deep Neural Network (DNN) inference, and achieves a 50x improvement per watt compared to conventional architectures. They have achieved a 50x performance improvement over general-purpose supercomputers for training DNNs as well [27].

These feats are a result of optimizing an architecture for a specific domain. A TPU operates on narrower words, and gets rid of caches and branch predictors, which better suits the requirements of its domain. The computation is done in a small number of large cores, consisting of a large number of individual FUs. This computation substrate, the heart of the TPU, is rooted in an architecture class called systolic arrays [28].

Systolic arrays consists of a grid of FUs that are interconnected by an internal network that can be fixed or programmable. Both FUs and interconnect are physically lightweight and fairly composable making them ideal building blocks for specialized accelerators. The execution model is simple; all FUs have their fixed operation, and whenever all its operands are ready it computes a result and dispatches it to the next node. This means that all parts of the array, from the initial inputs to the final outputs are pulsing with data. The findings in the late 70s and early 80s made it clear that this architecture made it possible to efficiently compute many basic matrix computations. In other words it is an ideal target for matrix-heavy applications such as DNNs.

Machine-learning DSAs has seen a surge in popularity in recent years, with efforts such as DC-CNN [29], and PuDianNao [30]. Research has targeted other domains as well; such as databases with the database processing units Q100 [31] and Widx [32], and graph processing with architectures such as the Graphicionado [33] and GraphPulse [34].

2.3 Reconfigurable Accelerators

Since its birth in 1960 [35], reconfigurable architectures has been used to specialize hardware to maximize efficiency for a particular program. This hardware is programmable and a change in a program is realized in hardware. Compared to ASICs and DSAs, they consist of reprogrammable logic at varying granularity, ranging from FPGAs that are reprogrammable at the bit-level to CGRAs that are reconfigurable at word-level. Another benefit is that reconfiguration enables us to repurpose chip area when we switch applications.

2.3.1 Field-Programmable Gate Array

The most flexible kind of reconfigurable architecture is the FPGA. Flexible in this context means that it can be programmed to perform any task, from regular computation to simulating complete processor cores fully asynchronously. They are ultimately programmed by specifying the circuit in Register-Transfer Level (RTL), which means that the programmer has complete control of the architecture at the bit-level.

Since this fabric is programmed with RTL it is not accessible to everyone, they are hard to program [36]. Recent efforts raising the programming level has resulted in High-Level Synthesis (HLS), an abstraction level that enables the programmer to write application code in C-like or C++-like languages that ultimately target an FPGA [37], such as Vivado HLS [38]. The main benefit of HLS is that it makes development easier and saves time when prototyping designs. For fitting problems HLS can be sufficient to achieve high performance, if carefully crafted. Producing production-grade FPGA configurations requires hardware knowledge and a substantial engineering effort.

In terms of efficiency, the main selling point of a FPGA is that it is cost efficient, compared to an ASIC realization. For certain applications that need a specially designed circuit, such as real-time applications, it is hard to beat an FPGA, where the alternative might be producing a small batch of ASICs. Compared to a DSA, already deployed FPGAs can be reprogrammed whenever there is a new algorithm or bug in the design [39]. Using an FPGA to prototype ASIC designs is also a major use case no other fabric is able to do well — simulating hardware in hardware.

The three main challenges when using FPGAs as accelerators are that (i) they run on comparatively low clock frequencies, (ii) reconfiguration is slow, and (iii) require high synthesis time.

FPGAs run on a very low **frequency**, low hundreds of MHz, depending on their current configuration, which means that they struggle to gain performance compared to other hardware architectures. The area overheads are also quite substantial, as there are a lot of interconnects, and the need for large amounts of multiplexing logic to enable reconfigurability of the network.

Reconfiguration is slow, currently requiring tens or hundreds of ms [40]. Partial reconfiguration is possible, requiring less configuration time to load configurations on parts of the FPGA. The programmer has to take care not to make programs that consists of multiple passes that uses multiple different configurations to produce results, as that will add substantial reconfiguration overhead. An alternative to reconfiguration is to acquire a larger FPGA.

The act of **synthesizing** the RTL design to a FPGA configuration, is time consuming, often requiring hours and days to complete. This rules out the possibility of an FPGA to adapt to runtime information, such as new acceleratable program regions or new applications. Whenever there is a possibility to dynamically extract parallelism in an already running application, the FPGA will remain idle. It is not well suited as a dynamically adapting accelerator target in its current form.

2.3.2 Coarse-Grain Reconfigurable Architecture

We have seen that systolic arrays have the possibility of reprogramming the internal network connecting FUs. A CGRA substitutes the fixed-function FUs with polyfunctional PEs, making the whole fabric reconfigurable. The effect is that the CGRA is reconfigurable in the spatial dimension, unlike the systolic fabric of the TPU.

A CGRA retains a lot of the flexibility we see in the FPGA, but sacrifices bit-level programmability to improve on all the drawbacks of the FPGA. These are also the key points that makes CGRAs attractive to pursue as an accelerator backend. Compared to the FPGA the CGRA can (i) run on higher clock frequencies, (ii) reconfigure in almost negligible amount of time, and (iii) require low synthesis time.

Attainable **frequency** of a CGRA is in the order of GHz, compared to a few hundred MHz in the FPGA. This directly equates to higher performance and promises of better speedups. If we assume that the power consumption is the same, this contributes to significant improvements in energy efficiency.

Reconfiguring the whole fabric is completed in 10-20 cycles. The low reconfiguration time overhead means that it is possible to reconfigure the fabric multiple times during a program's execution with negligible effects on runtime, effectively meaning little energy overhead. The effect is that more programs are tractable sources for targeting this kind of architecture. It even means that this architecture is truly reconfigurable in two dimensions, both the spatial dimension and the temporal dimension, compared to FPGAs that are really only reconfigurable in the spatial dimension due to its relatively high configuration overhead.

Fitting a program to statically scheduled CGRA consist of two major compiler steps: mapping and scheduling. This is analogous to the **synthesis** step for FPGAs, however comparably much simpler. Mapping is the action of distributing operations to PEs on the CGRA that support the functionality, that is decide a specific FU for a PE. If all operations in the DFG can be mapped, we can continue to decide a schedule. A static schedule is a configuration of the network on the CGRA that determines how operands flow between the mapped FUs. When operands travel through the fabric it is imperative that they match latencies and arrive at FUs at the same time, if not we will quickly reach a state of contention and halt acceleration execution. We can delay operands by e.g. inserting FIFO queues on the PE's input ports or send them on round-trips through the network.

The relevant search space for mapping and scheduling operations is relatively small compared to a comparably sized FPGA. While a CGRA that house 100 PEs require seconds to map and schedule most programs, an FPGA synthesis would require hours, if not days, to complete. This opens up arenas such as just-in-time compilation of schedules for some classes of programs [41].

PEs range from fully reconfigurable at every cycle, the operation it executes is decided when the data arrives, to reconfiguration of the substrate, or parts of the substrate. Drawing a parallel to CPUs this shares some characteristics with what we call dynamically scheduled and statically scheduled. In the case where a PE is dynamically scheduled, the operation is decided based on e.g. a tag that invokes the correct FU to operate on the operands that arrives at the inputs along with the tag. On the other hand, a statically scheduled PE does not have the circuitry to decide what operation is to be performed on its input operands, and will do the same operation on whatever inputs are served. A dynamically scheduled PE has area and power overhead compared to the lightweight statically scheduled PE.

A drawback to the flexibility CGRAs gain over systolic arrays is that each PE has to house multiple FUs in order to be reconfigurable, which incurs area overhead. This area overhead is not necessarily a problem, as we are seeing dark silicon all over the place and might even benefit from distributing the thermal budget over a larger area [6].

A disadvantage of CGRAs is that the communication network dictates the word width. This means that operating on anything more narrow than this wastes reconfigurable area. E.g. operating on single bits will result in terrible efficiency, wasting area and power. To some extent it is possible to pack operands into words and use the wide operations, but this is not feasible for all problems. Some architectures house FU variants that accepts subwords, so that two inputs of 64 bits can translate into e.g. 4 byte-operations or 2 2-byte operations, as with an adder that does 4 adds at the same time, consuming 8 bytes as input and produces 4 bytes as output [12]. This is essentially vectorization disguised in a single word.

Chapter 3

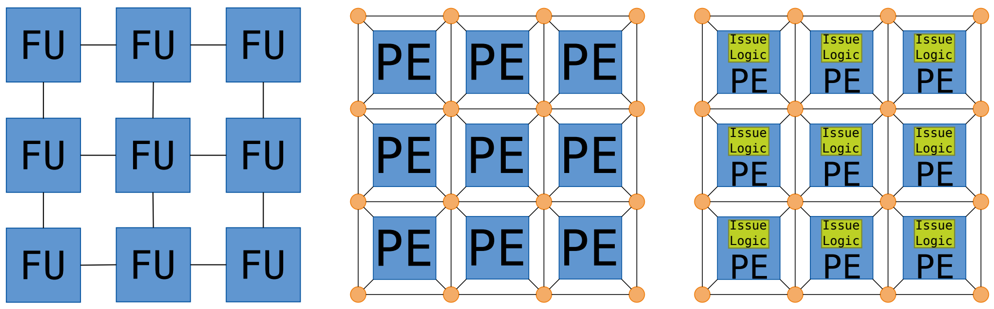
Architecting CGRA-based Accelerators

In this chapter we will introduce a set of architectures that is preliminary to the CGRA architecture we will use for experiments in Chapter 6. We start off by introducing systolic arrays, that enjoy high performance and high efficiency, trading off programmability. Then we describe static dataflow, which is very much like a programmable systolic array, at a marginal cost of efficiency. Finally we describe dynamic dataflow, that is dynamically scheduled, but less efficient than the former architectures.

To program these architectures we define a program and extract a DFG based on a set of operations that produce results. These operations have to be mapped to the PEs of the fabric. For systolic arrays, Figure 3.1a, this mapping is etched into silicon and can not be changed, we refer to a specific configuration of a PE as an FU. For the dataflow processors we can decide which FU (operation) a specific PE should assume; statically for static dataflow, see Figure 3.1b, and dynamically for dynamic dataflow, see Figure 3.1c. Systolic arrays and static dataflow has a strict schedule that perfectly match operand latencies, while the dynamic dataflow match operands dynamically and execute when two matching operands enter the PE. For systolic arrays this schedule is often etched in silicon and is not programmable.

3.1 Systolic Arrays

The preliminary architecture idea towards building CGRAs stems from systolic arrays, the invention by Kung and Leiserson, that enabled high-throughput acceleration by using parallel-performance geared network and processing structures [28]. It excelled in matrix-operation throughput, which has recently become important in speeding up DNN learning and inference. Especially so through Google's TPU. More recent Amazon started offering high-performance inference instances running on their in-house AWS Inferentia architecture, utilizing systolic arrays to



(a) Systolic array **(b)** Static dataflow **(c)** Dynamic dataflow

Figure 3.1: Grades of configurability from systolic array to dynamic dataflow. An FU houses a specific operation, a PE houses multiple FUs that can be configured statically for static dataflow, and dynamically for dynamic dataflow.

gain energy efficiency and higher performance [42].

A systolic array consists of FUs that have fixed functionality and operate in lock step. Each of these units can receive operands, compute a result, and emit a result each cycle. The specifics will vary between implementations, but the general idea is that data flows into the fabric and pulses through the array of FUs.

Kung and Leiserson mention that they reach optimal performance for matrix-multiplication with hexagonically shaped FUs. For matrix-vector operations multiplication they found that rectangular FUs fared better. This means that a single fabric will not be able to optimally accelerate every kind of application, even though one of the inputs are the same.

This also requires some changes to the internal network as well. Fully connected meshes will differ by the number of inputs and outputs. E.g. six connections for each FU when they are hexagonal and four connections when they are rectangular.

Even though the inherently fixed nature of systolic arrays makes them very inflexible, there exist non-pure systolic arrays, such as WARP [43] and its ancestors. These efforts introduced programmable networks to make the substrates more flexible and enabled them to be targets for multiple applications and domains.

The two main problems with systolic arrays are availability and usability. To our knowledge there are only two commercially available architectures, and these are just very recently become available to the public and are cloud-based instances offered by Google Cloud and AWS. As for programmability they have been hard to program in the general case, as the specific acceleration source has to be specifically optimized for the specific target systolic array.

We believe that the architectural idea of spatial computing is a candidate to reach further than matrix-operation acceleration. As both Google and Amazon have invested significant resources in development and large-scale development of machine-learning processors, this field in computing might get some needed traction outside of these domains as well.

3.2 Static Dataflow

Dataflow computers were extensively researched in 1970-1980, starting with static dataflow computers [44] [45]. They are programmed with data-flow languages that makes it possible to represent programs as graphs which are executed in a data-driven manner. This means that an operation fires when all input operands are ready, producing an output that is consumed by subsequent operations in the graph.

These DFGs are built as directed graphs that start with inputs and end with outputs. The nodes of a graph are operations, such as addition, multiplication and division. An example DFG is shown in Figure 4.3a. This graph has two inputs, a and b, that is the input for the addition operation, and c is the output of the graph. The graph is the program the dataflow computer executes. When inputs a and b are ready, they have values, they are sent to the adder and the sum is finally output from the program.

The mapping of this function to the dataflow substrate is in principle arbitrary. Any PE can compute the addition and will do so when two operands with a matching tag is ready. As an example, assume there is only one PE, then we can map the addition to that PE and feed it with data and expect outputs to be produced.

The supporting hardware requires scheduling of operations to be done statically, when compiling the program. The static dataflow computers does not have structures that support internal flow of control. This sounds a lot like the classic systolic array. We do sacrifice some efficiency and area to gain massive improvements in flexibility and ability to reuse the area for multiple types of applications. In Figure 3.1b we see that the area is small and we gladly accept the flexibility. Specifically, the modern DFG languages can be used to program both a systolic array and a static dataflow computer, but we can not reprogram the systolic array when it is already produced.

When we migrate this terminology, programming model and execution model to the CGRA domain, we will refer to this fully reconfigurable systolic array as a systolic CGRA. A systolic CGRA has no control capabilities in the PEs for maximum efficiency, and rely fully on a more complex network to route data as dictated by a statically determined schedule. Flexibility in the network can be achieved by e.g. lightweight switches between all PEs.

Scheduling of static dataflow is NP-hard and requires extensive use of heuristics [46]. When we have a DFG we need to map it to the available PEs, route data between them and ensure that we time the values to arrive at PEs at the same time. If the fabric does not house enough PEs required by our DFG, we can not map the program. If we can map the program, but not transmit the data between the PEs, we can not route the program. If we can route, but not make operands arrive at PEs at the same time, we can not schedule efficiently and will quickly have to deal with network contention.

3.3 Dynamic Dataflow

We can enjoy very high efficiency and very high performance on static dataflow for amenable workloads. We can not, however, expect to run anything more exotic on a static dataflow computer than we can on a systolic array. A natural evolution of the static dataflow computer was adding dynamic scheduling capabilities to achieve more general parallel-compute fabrics, see Figure 3.1c. This led to dynamic dataflow computers, such as the Manchester Dataflow Computer [47].

Another important aspect of dynamic over static dataflow computers was that they could handle multiple execution contexts in the form of reentrant code as they could tag operands and differentiate between owning processes, meaning that multiple programs could use the dataflow unit with the same configuration at the same time without interfering with each other.

The execution process of a real tagged-dataflow architecture is quite involved and the MIT tagged-token architecture is thoroughly explained in [48]. Here we give a brief overview of a basic dynamic dataflow computer. A network connects PEs to facilitate communication between the nodes. A common approach to achieve the dynamic aspect is to attribute input data with a tag. This tag is used to match operands, i.e. when an operand enters a PE it waits for another operand with a matching tag before the operation they are attached to executes. The tag also matches an operation, which decides the operation to be executed on the operands.

On its own it sounds like dynamic dataflow only contributes hardware complexity on top of static dataflow. However, with dynamic dataflow we are now able to accelerate code that inhibits parallelism, but does not have regular memory accesses or data-dependent loop-counter variables. An example is Code listing 2.3, This is not possible to compute efficiently, or sometimes at all, in pure static dataflow as we do not have information about i before we compile the program and can not infer j to efficiently decide on a good vectorizing scheme to account for the triangular stream pattern. With dynamic dataflow we also get dynamic scheduling which lets us compute these workloads.

More support in the hardware makes it easier to schedule applications, at the cost of area and increased hardware complexity. This hurts the efficiency of programs that do not need this capability. In the next chapter we look into a state-of-the-art CGRA-based accelerator and how it balances this tradeoff.

Chapter 4

REVEL: A CGRA Accelerator Case Study

The REVEL architecture aims to replace multiple ASICs that target the IoT and 5G workloads [12]. The architecture combines both static dataflow and dynamic dataflow, specifically a tagged dataflow. This hybrid CGRA has coined the terms *dedicated region* for the reconfigurable static-dataflow region and *temporal region* for the dynamic-dataflow region. A combined architecture that can house highly-optimized vectorized code in the dedicated region and keep non-critical code running in the temporal region.

We define non-critical code as code that is not on the critical path, i.e. not important for overall throughput of parallel execution. Two examples are high-latency operations and high-latency sequences of code in the outer loop, such as the square root operation in the outer loop of Code listing 2.3. The outer loop can be scheduled entirely to the temporal region to execute concurrently, while the inner loop can be vectorized and be scheduled to the dedicated region for maximum throughput and efficiency.

4.1 Stream-Dataflow Architecture

Stream-Dataflow Architecture (SDA) is an architecture designed from the ground up [11], based on the common efficiency features of state-of-the-art DSAs [15], we introduced in Section 2.2.2; concurrency, computation, communication, data reuse and coordination. This architecture is the base of Softbrain, an implementation that uses a CGRA as its reconfigurable substrate. The CGRA is an entirely dedicated region.

The novelty of SDA is that the architecture combines a programmable stream interface with a reconfigurable computation substrate. To execute programs, it uses a light-weight control core to interpret a program that dispatches commands to streaming units. The streaming units are responsible for moving data between memory units and the CGRA. The memory units are the memory system, an

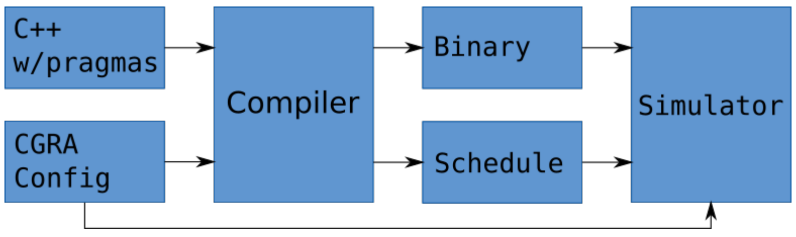


Figure 4.1: REVEL compiler toolchain. A C++ w/pragmas program and a CGRA configuration is the input to the compiler. The compiler produce a binary the control core executes, and a schedule to configure the CGRA. The simulator loads the CGRA configuration and executes the binary, which loads required schedules on demand.

architecture-local scratchpad, and a recurrence stream that feeds data from the CGRA outputs to its inputs.

The word width is 64 bits, and the network and all PEs are optimized to read 64 bits every cycle. There is a vector-port interface surrounding the CGRA. It accepts multiple memory widths, allowing us to communicate up to 512 bits per vector port, in a granularity of up to 8. This means we can vectorize memory access and read 8 64-bit operands.

Compared to DSAs the SDA is more general in that it has a reconfigurable datapath and reconfigurable memory streams. While a DSA is the target for a single domain, the SDA can in principle be a target for multiple domains. We see the SDA as a generalized DSA.

The SDA achieves concurrency by spatially distributing work, e.g. by unrolling multiple inner-loop pipelines. The CGRA houses specialized FUs inside the PEs to compute more efficiently. Communication inside the CGRA is efficient, and streaming data to and from memory overlaps IO and computation, transforming them to concurrent actions. Recurrence streams enables immediate data reuse, and a scratchpad eliminates most off-chip memory accesses related to data reuse with a larger reuse distance. Finally, coordination is specialized by having a static schedule when computing on the CGRA.

4.2 Reconfigurable Vector Lanes (REVEL)

The REVEL architecture builds on SDA, and introduces a programming model and execution model that combines two types of PEs on the same fabric, as illustrated in Figure 4.2. The figure illustrates the complete concept, with multiple lanes of execution. We consider only on lane in this work, and we refer to Vector Lane 1 in the figure as the CGRA. Hence, there is no shared scratchpad or XFER elements that would otherwise act as lane-to-lane communication facilitators, and the vector-stream control core produce commands for only Vector Lane 1.

Pink nodes in the DFGs are targeting the dedicated region, the pink area in the CGRA. Complementary the purple DFG nodes represent nodes that target the tem-

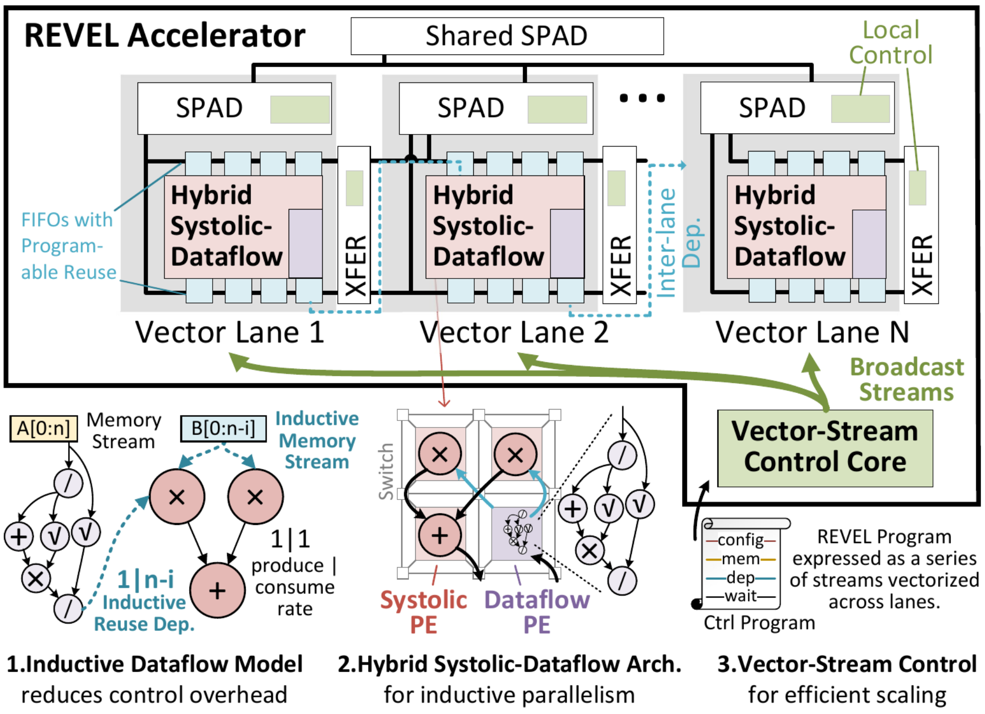

REVEL Accelerator
Shared SPAD
Local Control
SPAD
SPAD
SPAD
FIFOs with Programmable Reuse
Hybrid Systolic-Dataflow
XFER
Inter-lane Dep.
Vector Lane 1
Vector Lane 2
Vector Lane N
Broadcast Streams
Vector-Stream Control Core
A[0:n]
Memory Stream
B[0:n-i]
Inductive Memory Stream
1|1 produce | consume rate
1|n-i Inductive Reuse Dep.
Switch
Systolic PE
Dataflow PE
config
mem
dep
wait
Ctrl Program
REVEL Program expressed as a series of streams vectorized across lanes.
1.Inductive Dataflow Model reduces control overhead
2.Hybrid Systolic-Dataflow Arch. for inductive parallelism
3.Vector-Stream Control for efficient scaling

Figure 4.2: The REVEL architecture. The figure is reproduced from [12].

poral region, the purple region in the CGRA. In Figure 4.2, 1. Inductive Dataflow Model shows how a multi-DFG configuration can be modeled. Here, for each element the temporal region produce, the dedicated region will consume that element $n-i$ times.

In 2. Hybrid Systolic-Dataflow Arch. we observe that the dedicated region is statically mapped with one FU per PE. Since the CGRA is small and house only one temporal PE, all of the temporal operations are mapped to a single PE that dynamically schedules its execution. 3. Vector-Stream Control is not relevant, as we do not vectorize streams across lanes. The architecture do vectorize streams for each CGRA, so we still benefit from vectorized streams.

The idea is that more programs can be accelerated efficiently on a hybrid architecture than on purely dedicated CGRA. The REVEL architecture lends itself to distributing work to multiple, similar CGRAs. In this work we will not explore this ability, we are only interested in single-fabric mapping and scheduling.

The major benefit from adding a temporal region is that the mapping is easier, the critical paths in the computation can be mapped to the highly efficient dedicated region of the CGRA, while we can map the non-critical parts to the temporal region. An example is a long-running (chain of) code in the outer loop, such as for the benchmark qr_q_D that has multiple divisions and square root operations in the outer loop. This benchmark also has the inductive dependency property, where the induction variable of the inner loop is dependent on the outer loop's

Instruction	Bitwidth	#Operands	#Outputs	Latency	Throughput
Add8	8	2	1	1	1
Add16	16	2	1	1	1
Mul16	16	2	1	1	1
Add32	32	2	1	1	1
Mul32	32	2	1	1	1
Add64	64	2	1	1	1
Mul64	64	2	1	1	1
Add16x4	64	2	1	1	1
Mul16x4	64	2	1	1	1
Add32x2	64	2	1	1	1
Mul32x2	64	2	1	1	1

Table 4.1: A sample of operations that can be allocated to a PE in REVEL's reconfigurable substrate

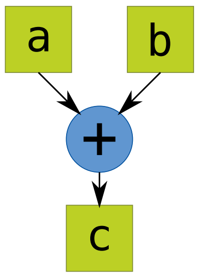
induction variable, see Code listing 2.3 for an example.

The programs we run on REVEL are compiled by the steps shown in Figure 4.1. The compiler toolchain takes two inputs, the source code of the program we want to accelerate and a CGRA configuration. As noted in Section 2.1.2, we program in C++ with #pragma directives to annotate where we want streams and generation of an offloadable accelerator (schedule) and a binary that the control core will execute. Currently all execution is simulator based.

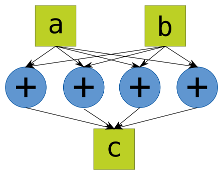
As an example we briefly describe the #pragma directives, and process of compiling the dot-vector function in Code listing 2.2. `#pragma ss config` on line 3, dictates that the following block is offloadable. `#pragma ss stream` on line 6, is a directive to allocate a memory stream based on the following loop header. `#pragma ss dfg dedicated unroll(4)` on line 7, asks the compiler to analyze the following loop for dependencies and build a DFG of the contents, if possible, and makes sure to unroll the pipeline four times. The compiler then generates a mapping and schedule of the DFG, given the CGRA configuration.

REVEL's reconfigurable substrate is coarse. The narrowest data element is 8 bits and widest is 64 bits. The memory interface and the internal network is optimized for communicating and computing on 64-bit operands. A high grade of utilization is reached when the width of both the network and currently configured FU is saturated by data traveling through the substrate.

Most operations that can be allocated to a PE is capable of operating on multiple 8-bit, 16-bit and 32-bit operands at the same time. All communication from PE to PE is 64-bit wide, meaning that consolidating multiple smaller operands in one package results in better utilization of the internal network. A non-exhaustive list in Table 4.1 illustrates a few combinations.



(a) Single pipeline Dataflow Graph.



(b) Unrolled Dataflow Graph.

Figure 4.3: Difference between a single pipeline and unrolling the vector addition loop by a factor of four.

Add8 is the simplest function and adds two 8-bit operands. This operation is not very efficient, as it only operates on 1/8 of the available internal-network bandwidth. Allocating an `Add8` to a PE should be done only when necessary. `Add16` and `Mul16` takes two 16-bit operands and is slightly more efficient than its 8-bit counterpart. Then comes 32-bit versions and lastly 64-bit versions. 64-bit versions are always most efficient in terms of communication.

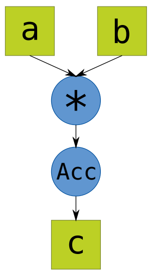
To overcome inefficiencies when computing with narrow operands, there exist operations that can consolidate computation of multiple narrow operands at the same time. `Add16x4` and `Mul16x4` can calculate four 16-bit operations at the same time, in the same PE. This results in complete utilization of the network to and from that PE. The same goes for `Add32x2` and `Mul32x2`, computing two 32-bit operations. The last four operations has two 64-bit inputs, that is distributed to multiple binary operations operating on two 16-bit or 32-bit operands. The outputs are 64-bit wide, keeping the order of the incoming operands. This resembles operation vectorization, albeit implicit.

We illustrate the DFG of the vector addition function from Code listing 2.1 in Figure 4.3. In Figure 4.3a we have a single pipeline, which means we have no explicit parallelism. To express parallelism we unroll the pipeline four times and end up with Figure 4.3b. We see that we can easily extract parallelism from vector addition by arbitrary unrolling.

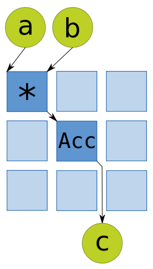
4.3 Putting it All Together: Accelerating a Dot Product Kernel

In this example we will show steps on how to map the dot product function in Code listing 2.2 to the dedicated region of a sample REVEL instance. First we show the naive mapping of a single iteration. Then we expand the DFG by unrolling the computation to increase performance. To increase efficiency we add reduction of partial sums to the DFG, before we finish with vectorized data streams.

Without the loss of generality, We assume that the REVEL instance is organized as a grid with a total of 9 PEs that can be configured to any form of FU. Accounting

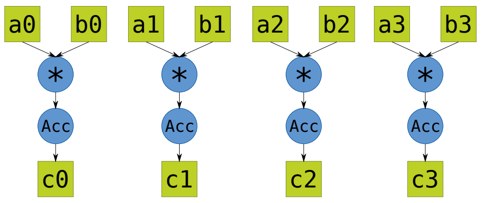


(a) Dataflow Graph.

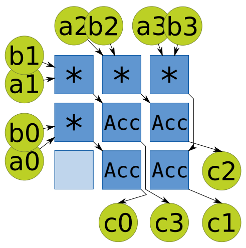


(b) Mapping.

Figure 4.4: Naive vector dot product. Targeting a 9-element CGRA.

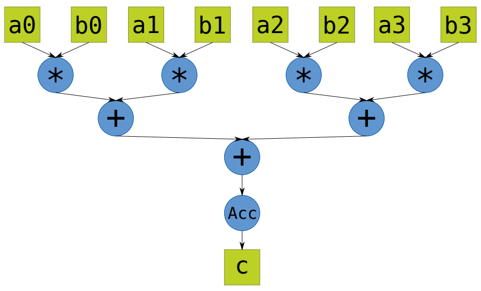


(a) Dataflow Graph.

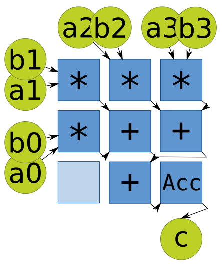


(b) Mapping.

Figure 4.5: Unrolled vector dot product. Targeting a 9-element CGRA. Unrolling factor is 4 to fill the CGRA.

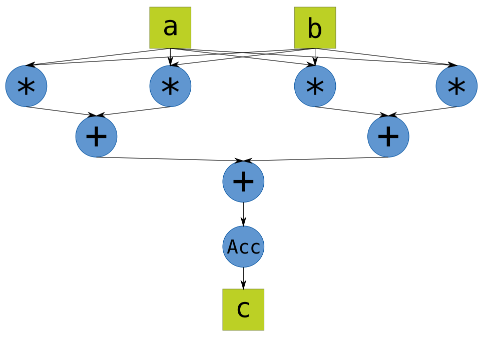


(a) Dataflow Graph.

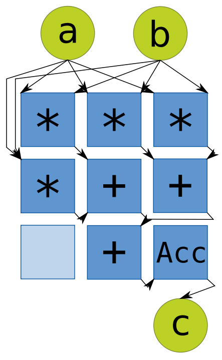


(b) Mapping.

Figure 4.6: Unrolled and reduced vector dot product. Targeting a 9-element CGRA.



(a) Dataflow Graph.



(b) Mapping.

Figure 4.7: Unrolled, reduced and memory-interface vectorized vector dot product.

for limitation in what can be mapped to a PE makes mapping more complicated in practice, but does not change the underlying principle. The PEs can receive data from any direction, but can only deliver data down to the right (south-east).

Naive mapping: Figure 4.4b shows the naive mapping of the DFG in Figure 4.4a. There are no surprises here, we stream input data to the input ports, calculate a partial sum and accumulate the result before releasing the result when the input stream is exhausted. To successfully compute the dot product we need to invoke a CGRA instance as many times we read input data from memory, totaling N times.

Unrolling: Figure 4.4b illustrates that the naive approach poorly utilizes the available resources on the CGRA. With unlimited resources we could unroll this DFG indefinitely. We are limited to a unrolling the loop by a factor of 4, since the CGRA has only 9 PEs. The resulting DFG and mapping is showed in Figure 4.5b and Figure 4.5a, respectively. By unrolling we improve performance of the dot product computation by 4x, reducing computation time and number of invocations to $N/4$ compared to the naive approach.

Reduction: We observe that the results of the unrolled computations are written to memory, and that summing these partial results have to be done in the CPU, or in another CGRA instance that handles reduction only. In any case we have to communicate the results to the memory system, but we can do better. By introducing reduction in the DFG, Figure 4.6a we can retain the state of the partial sums and total accumulation inside the CGRA, Figure 4.6b. This means we read $N * 2$ input operands, N from a and N from b, and write to the memory system only once, the value of dot, when the complete dot product is produced. We gain efficiency by moving computation nodes from the CPU to the DFG.

Vectorization: To squeeze even more efficiency from the system we turn our attention to the memory interface. Until now we have read data one operand at a time for each of the input ports. To do this predictably we had to partition the memory reads over, possibly, non-contiguous memory. Considering access patterns that can affect the memory system efficiency and performance, we can not guarantee efficiency without vectorizing memory access. The underlying memory system, cache-block size, eviction policy, memory banks, etc. is out of our control and we can not rely on the memory system to handle memory efficiently [49].

Therefore we choose to consolidate memory accesses by streaming all input data contiguously to two vector input ports that will handle distribution of the operands to the correct FUs — for this example this means that the contiguous memory for both vectors a and b are streamed to wide-vector input ports of the CGRA. The resulting DFG, Figure 4.7a, shows a single entry point for each of the inputs a and b. The output is not vectorized, because acc is only one data element. We see that the compute units are mapped exactly as before, so the only difference is that inputs are delivered from two 4-operand-wide input ports instead of eight single-operand ports, as shown in Figure 4.7b.

Chapter 5

Experimental Setup

5.1 Simulator

We use a modified gem5 [51] provided by the PolyArch Research Group at UCLA, [12], that is available at [13]. All configuration sizes, NxN, are modeled with a $0x0, 1x1, ..., nxn$ temporal region located at the bottom right of the configuration. From now on we will refer to a CGRA size as the letter c followed by a number, e.g. c5 will refer to a CGRA of size 5x5. We abbreviate temporal regions in the same way, with the letter t, e.g. t3 represents a temporal region of size 3x3. Combining the two we get e.g. s8-t2, meaning a CGRA of size 8x8 housing a temporal region of size 2x2. In Table 5.2 we have listed the number of PEs for all configuration sizes. All variants of the configuration size, i.e. different temporal-region size, has the same spatial distribution of PEs.

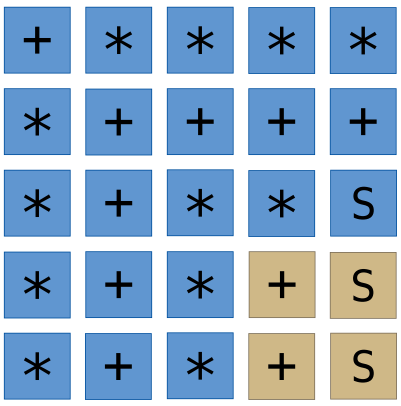
We model multiple CGRA configurations, ranging from size 5x5 up to 10x10. All configurations are quadratic and an increase in the size appends PEs at the right edge to the right, and PEs on the bottom edge downwards, as illustrated in Figure 5.1. s5-t2 in Figure 5.1a is a CGRA of size 5 with a temporal region of size 2, and Figure 5.1b shows the s6-t2 configuration, with arrows to show

Parameter(s)		**Value(s)**
Vector ports	Width	2x512, 2x256, 1x128, 1x64 bit
	Depth	4-entry FIFO
Scratchpad	Size	8 kB
	Bandwidth	512 bits/cycle
Number of lanes		1
Control core	RISCV [50], 5-stage, single-issue 16kb d$, modified w/stream-command instructions	
System cache	L2 cache	2048 kB

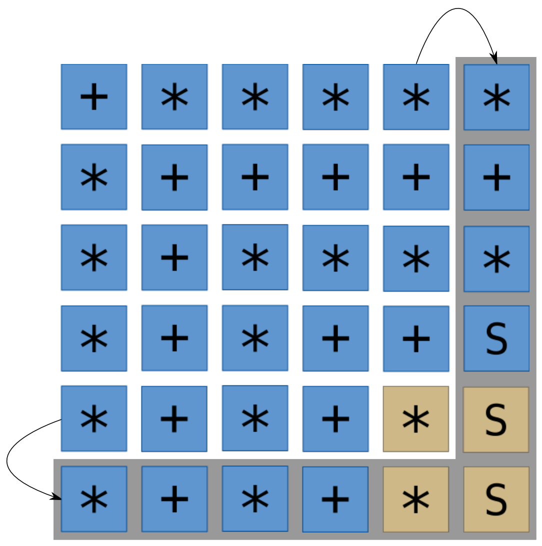
Table 5.1: Simulator Setup

Configuration	#Add	#Mul	#sqrt/div
s5	10	12	3
s6	14	19	3
s7	21	25	3
s8	27	34	3
s9	36	42	3
s10	44	53	3

Table 5.2: Number of PEs per CGRA configuration



(a) CGRA configuration s5-t2, blue PEs are dedicated and brown are temporal. +: add, *: multiply, S: sqrt/div.



(b) CGRA configuration s6-t2, blue PEs are dedicated and brown are temporal. The grey area illustrates the expansion over s5-t2. +: add, *: multiply, S: sqrt/div.

Figure 5.1: CGRA configurations and how they grow. They grow to the right and downwards, replicating the add and multiply PEs. The sqrt/div PEs are always situated bottom-right.

PolyBench	**MachSuite**	**DsP**
2mm	crs	fft
3mm	ellpack	qr_r
atax	gemm	qr_q
gemm	md	mm
gemver	stencil-2d	
gesummv	stencil-3d	
mvt		

Table 5.3: Benchmarks

direction of replication, while the gray area shows the difference between the two configurations. Note that the three sqrt/div PEs stick to the bottom-right corner of the configuration. To enable consistent scaling to larger CGRA sizes we designed a basis layout in s5 that allows for the described growth pattern.

5.2 Benchmarks

We have listed the benchmarks that are run on the simulator in Table 5.3. Some of the benchmarks in the different suites have the same names. To distinguish between these, and make it easier to reference a particular benchmark from a specific benchmark suite, we add a prefix consisting of an underscore followed by an uppercase letter, the first letter in the benchmarks suite: _M for MachSuite, _P for PolyBench and _D for DsP. E.g. 2mm_P is the benchmark 2mm from the benchmark suite PolyBench.

In this collection of benchmarks there are mostly programs that normally benefit from regular parallelization, such as matrix multiplication and stencil applications: 2mm_P, 3mm_P, atax_P, gemm_P, gemver_P, gesummv_P, mm_D, mvt_D, stencil-2d_M, stencil-3d_M. They are particularly suited for streaming and the computation regions are typically easy to spatially distribute.

The DsP suite is a collection of benchmarks crafted by the PolyArch group targeting their REVEL paper [12]. In this suite one benchmarks, qr_q_D, specifically target the temporal region. The benchmarks are collected from the compiled-workload collection at [13] and are mostly untouched. gemm_M is slightly modified to make it produce correct results, see Appendix A for the modified implementation.

5.3 Metrics

The simulator can report metrics for a specific run-time region. This region of interest is programmatically set to enclose the computational kernels so that we measure only the CGRA-specific parts of the benchmarks, leaving general system overheads out of the data collection. We report the number of cycles needed to execute the region of interest.

As a baseline we choose the baseline CGRA configuration of Weng et al. [12], namely the `revel-1x1` configuration (our `s5-t1` configuration is a slight variation of the baseline that enables more consistent scaling to larger CGRA sizes). To compare across different sizes and across benchmarks we chose to normalize all data to the baseline configuration.

Chapter 6

Results

In this chapter we quantitatively analyze the CGRA performance to complement and validate the qualitative analysis in Chapters 2 to 4. More specifically, we first examine the performance of purely dedicated CGRAs, i.e. all reconfigurability is mapped and scheduled statically. Then we introduce a temporal region to identify potential gains in performance and mappability.

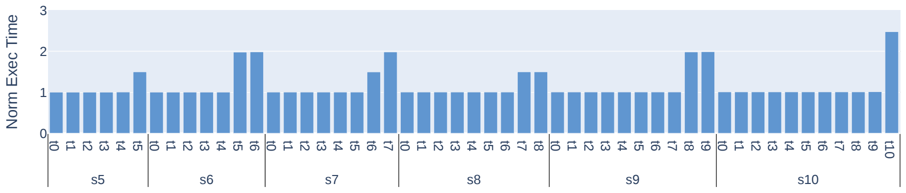
6.1 Performance versus CGRA Size

Figure 6.1, 6.2, and 6.3 show execution time for the region of interest normalised to the REVEL baseline across the PolyBench, MachSuite, and DsP benchmarks suites, respectively. The REVEL baseline is not shown in the plot as its normalized performance is always one.

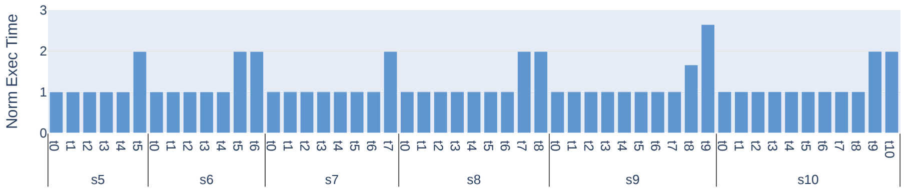
Benchmarks that normally benefit from regular parallelization[1], such as matrix multiplication and stencil applications show that the performance is virtually the same for all configuration sizes. A DFG will map to a CGRA whenever there are sufficient computational resources on the fabric. All of our results shows that there are no significant gains in increasing the number of mappable PEs by increasing the size of the reconfigurable fabric. This could mean that the memory system is operating on its limit, providing maximum bandwidth. This is, however, unlikely, as all benchmarks do not share memory access pattern.

From the perspective of mappable CGRAs it is evident that as long as a schedule can be found for a DFG, the same schedule latency is attainable regardless of CGRA size. This is also the case for benchmarks that require frequent reconfiguration, such as fft_D, incurring higher total configuration, fill and drain overhead. The overheads of filling and draining the pipelines are the same for all configurations meaning that there is no difference in schedule latency. If not, there would be aggregate differences for larger CGRAs when we could have made schedules with lower latencies.

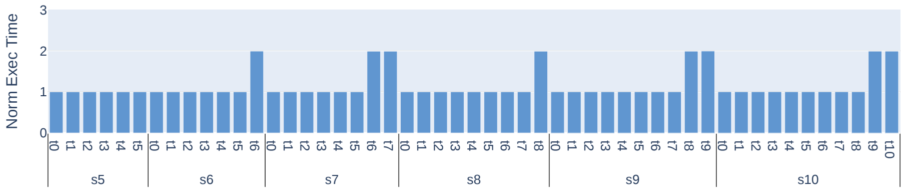
[1]2mm_P, 3mm_P, atax_P, gemm_P, gemver_P, gesummv_P, mvt_P, stencil-2d_M, stencil-3d_M, and mm_D in Figure 6.1a, 6.1b, 6.1c, 6.1d, 6.1e, 6.1f, 6.1g, 6.2e, 6.2f, and 6.3b, respectively



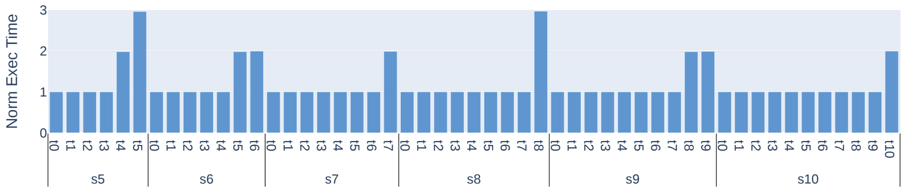
(a) 2mm_P



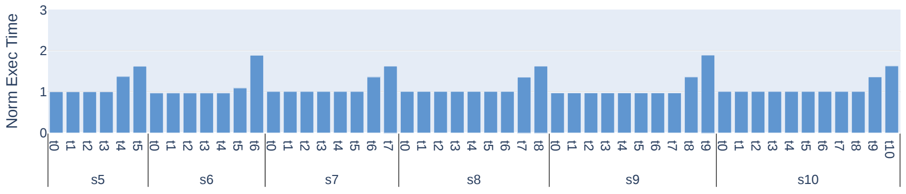
(b) 3mm_P



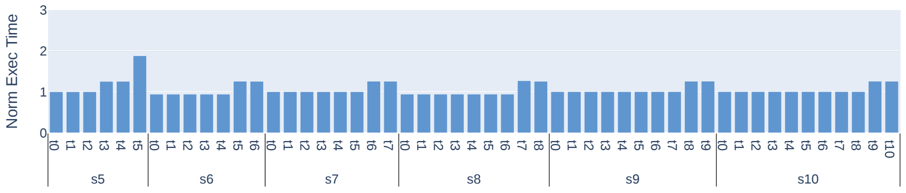
(c) atax_P



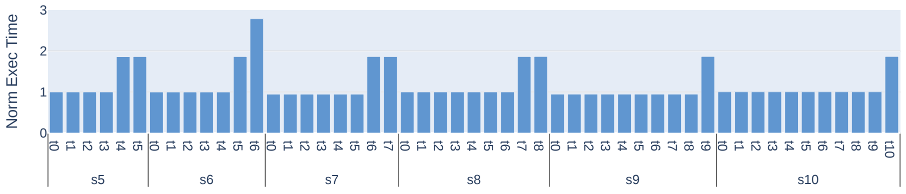
(d) gemm_P



(e) gemver_P

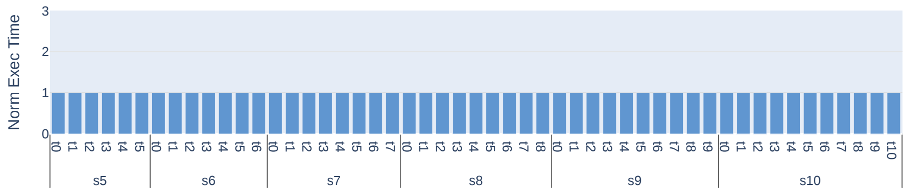


(f) gesummv_P

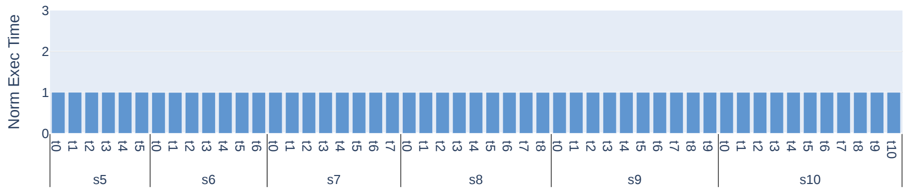


(g) mvt_P

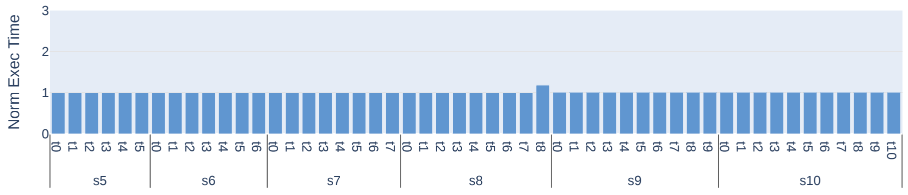
Figure 6.1: Benchmarks from the PolyBench suite



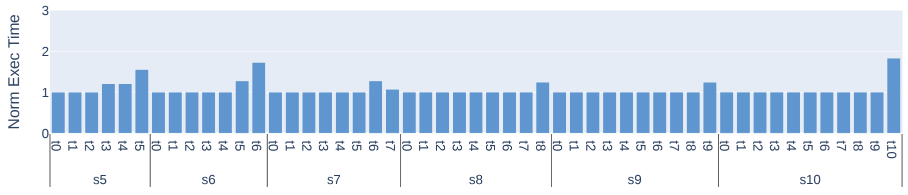
(a) crs_M



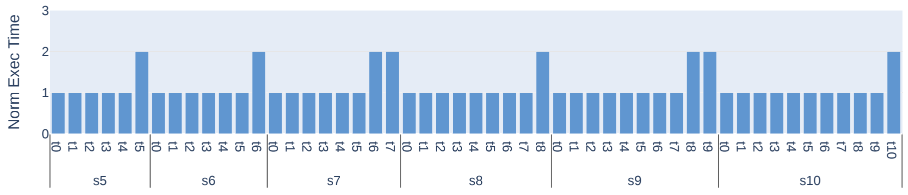
(b) ellpack_M



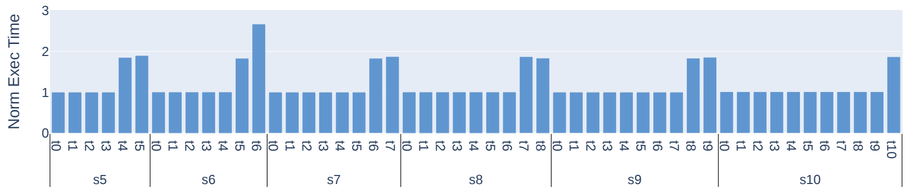
(c) gemm_M



(d) md_M

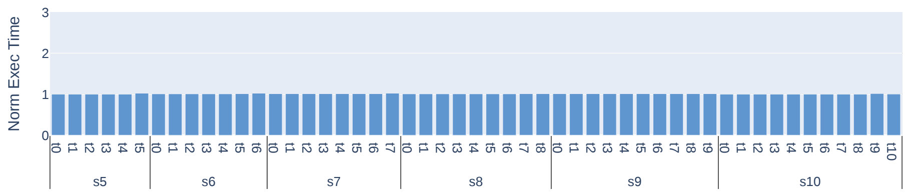


(e) stencil-2d_M

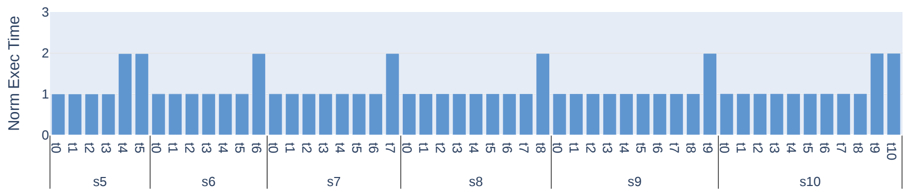


(f) stencil-3d_M

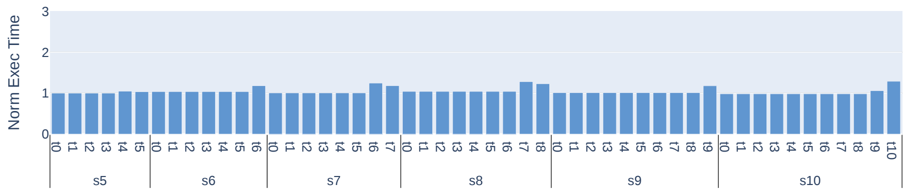
Figure 6.2: Benchmarks from the MachSuite suite



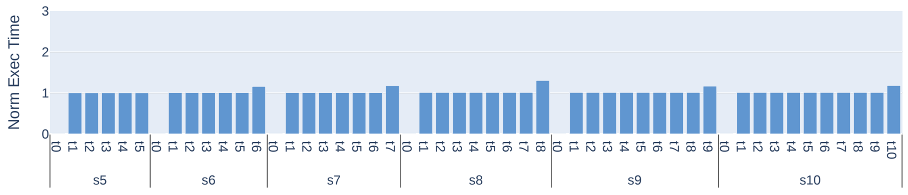
(a) fft_D



(b) mm_D



(c) qr_r_D



(d) qr_q_D

Figure 6.3: Benchmarks from the DsP suite

For larger CGRAs, reconfiguration overheads could be mitigated by swapping in the required mapping in other parts of the fabric, i.e. partial reconfiguration. Otherwise we will not be able to execute multi-pass programs on a static dataflow without writing large amounts to memory between passes, or carefully design data streams so that we can store parts of the data in the scratchpad.

6.2 Performance versus Temporal Region Size

We now turn our attention to how performance evolves when the temporal region varies. All benchmarks are run with a temporal region of size 1, and every increase in size until the maximum size for each configuration. E.g. for a CGRA of size 5 (s5) we run all benchmarks on configurations s5-t1, s5-t2, s5-t3, s5-t4, and s5-t5.

We find that none of the benchmarks increase their performance when the temporal region increases in size. This is not surprising on its own, if it were not for the fact that performance starts to degrade when the ratio of temporal-to-dedicated-region increases.

Regions that are not explicitly marked temporal can be mapped to the temporal region when the dedicated region is too small to house the complete DFG. This sometimes results in that the mapping algorithm chooses to overprovision PEs with multiple FUs nodes (operations in the DFG). For small fabrics where resources are scarce, this is a viable option to keep data in the fabric to reduce communication overhead, and possibly achieve higher efficiency than a GPP alternative. For fabrics that would otherwise be able to spatially distribute all operations to the temporal region we would expect to achieve the same performance as a comparably sized dedicated region.

When the temporal region becomes large, or the configuration is entirely temporal, performance, surprisingly, tends to degrade by a factor of about 2 to 3. To illustrate why this is the case, we examine `gemm_P`. For `s5-t0`, `s5-t1`, `s5-t2` and `s5-t3` the performance is the same as the baseline. There are two DFGs running at different times in this benchmark, they are swapped in when needed, and the one with the most nodes has a total of 9 nodes: 5 multipliers and 4 adders. This DFG maps to a total of 9 FUs on the CGRA. The number of available temporal PEs eats into the number of available dedicated PEs and the total number of dedicated PEs in the `s5-t3` configuration is $5 * 5 - 3 * 3 = 16$. We observe that the DFG is still spatially distributed to the dedicated region.

However, when the temporal region grows more, we observe that the performance degrades. The `s5-t4` consists of $5 * 5 - 4 * 4 = 9$ dedicated PEs, of which 8 are multipliers and 1 is an adder. We see that we are missing 3 adders to map this entirely to the dedicated region and the mapper chooses to allocate temporal PEs to supply enough resources to house all FUs.

In these cases the nodes in the DFG is distributed over the temporal region, and we expect to achieve the same performance as with a pure dedicated region, at the cost of increased power dissipation. However, we are not seeing this. When inspecting the final map and schedule, we observe that two adders are allocated to the same temporal PE. This will serialize execution and double total runtime, which is what we observe in Figure 6.1d. The same is true for `s5-t5`, the final map and schedule has allocated three multiplications to a single PE, which again leads to serialization and a trifold in execution time compared to an optimal schedule.

Sometimes, overprovisioning a temporal PE can be beneficial when it will produce more output data than subsequent stages are able to consume in the same amount of time. It can be used as a means to delay the operands, which otherwise would be realized in FIFO queues or detours in the CGRA network. Computation requires less energy than moving data around, so this might even offset the negatives of temporal execution.

For benchmarks that have DFGs that fit in the dedicated region alone, we find no benefits when adding a temporal region. In some cases it can free up resources in terms of PEs, but we have to remember that a dataflow PE is 5.8x the area of a systolic one. Hence, the overheads of the dataflow PE outweighs their benefits in this case.

By adding a temporal region we gain the ability to map a wider set of problems to the reconfigurable fabric, at the cost of higher power consumption and larger area footprint. Given that we can reduce or get rid of external communication, roundtrips to memory and fallbacks to a GPP, these tradeoffs are worth it.

6.3 Explicit Temporal Region

For `qr_q_D`, Figure 6.3c, the only benchmark that have a code region that is programmatically specified to require a temporal region in order to map to the architecture, we observe that it cannot map to the purely dedicated configurations, even for `c10-t0`.

For `qr_q_D`, one reason is that there are four division operations and two square root operations in one of the DFGs. That requires at least six PEs supporting those operations, while all our configurations only house 3. In the temporal region this is handled by dynamic scheduling and temporal sharing of resources, but the dedicated units do not have this ability, hence the lack of mappability.

Chapter 7

Towards Scalable CGRA Accelerators

Chapter 6 shows that the current state-of-the-art approaches do not scale to larger CGRA architectures. While limitations of the simulator infrastructure meant that we could not empirically validate improvements, we have attained a better understanding on important aspects to consider how to achieve scalable CGRA accelerator performance — and the aim of this chapter is to make these observations explicit.

We illustrate in Figure 7.1 the important aspects that drive performance and efficiency when accounting for the interplay between application, and mapping and scheduling, and the architecture. We need to consider limitations and opportunities between mapper/scheduler and architecture, (3) and (4) in the figure, and the common limitations and opportunities between the toolchain and the application, (1) and (2) in the figure.

Considering that the programming interface is similar to other parallel programming approaches, we are confident that we are able to express sufficient parallelism in the application and communicate it to the toolchain, (1) in the figure. However, when we try to unroll and vectorize benchmarks in excess of the original factors we find in the benchmarks, we fail to produce results with the toolchain and simulator.

Specifically, we wanted to examine benchmarks that theoretically have no computational limits in parallelism, such as `mm_D`, to explore memory bandwidth limitations (the memory wall). We expected that, as we had expressed a high level of parallelism in (1), that we would enjoy optimizations at the framework level for increased efficiency. We encountered problems as a combination of limitations in the scheduler and in the CGRA configurations. To begin with we tried to naively unroll by larger factors, i.e. express even more parallelism (1), but scheduling did not succeed. Then, we added more vector ports to the configuration, addressing shortcomings in number of vector ports (4), to allow the scheduler more freedom in where to direct the data streams. We had no luck. Finally, we increased the size of the CGRA configuration (4), but the scheduler was not able to find a viable

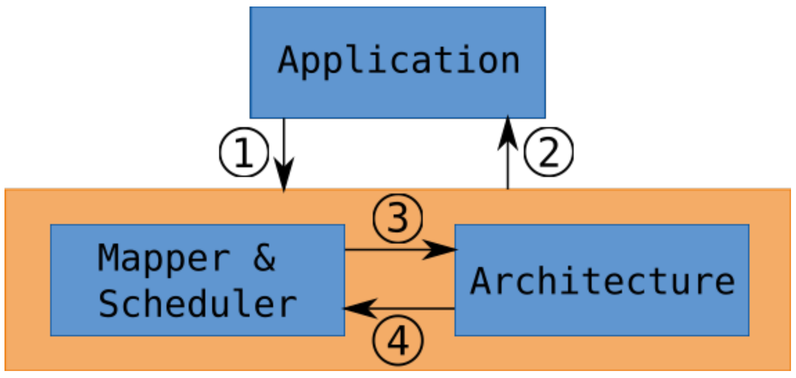


Figure 7.1: Drivers of performance and efficiency in the application vs mapping/scheduling and architecture.

schedule with more degrees of freedom.

What should have been opportunities in point (4) materialized as limitations when they arrived at the mapper and scheduler stage. We realize that a larger CGRA might increase the search space of the mapper and scheduler, but even DFGs that saturate the CGRA at smaller sizes suffer. It is likely that vector ports are the main suspect, as it is this interface that most often hinders successfully mapping and scheduling an application.

It turns out that the heuristics used to map and schedule a DFG to the CGRA are insufficient to map anything that is wider (higher unrolling factor) than what can be accommodated by two 8-operand vector ports. When we try to schedule more than 16 vectorized input operands in total, the scheduler struggles to find a schedule, and ultimately fails. One could assert that this is a matter of letting the scheduler run for longer, but we are confident that stopping the scheduler after a week's worth of work without success is sufficient evidence that it will not find a schedule. At least when compared to the normal schedules, which it decides on in a matter of seconds.

To elaborate on (3), we have to look to our results from Chapter 6 to see opportunities and limitations. If the CGRA is entirely dedicated, the scheduler has to decide on a static schedule. If portions of the CGRA is temporal, the mapper and scheduler has to figure out whether the application can map to the dedicated region or if it has to rely on dynamic scheduling capabilities of the temporal region to offload the program. First, an edge case is for the scheduler to move all scheduling to runtime and communicate to the mapper that it can map the DFG to the CGRA at free will. The worst case being that it maps all operations to a single temporal PE. This counters intuition, however a similar approach is sometimes taken when the dedicated region lacks resources and parts of a DFG has to be mapped to the temporal region. We would expect that the mapper and scheduler iteratively improved on each other to attain high performance.

If an application does not inhibit characteristics of a typically acceleratable program, as described in Section 2.1.1, we expect (1) to be a limitation. In these cases we should not strive to compute the application on the CGRA. If the algorithm can be rewritten to host these characteristics it is a viable candidate. Care have to be taken not to exclude applications based on a first-sight basis. At least currently, the programmer plays the sole role of application analysis. As for (2) what we probably will encounter is classes of problems that should be able to attain performance on this architecture, but insufficient mapping and scheduling algorithms, limits how far we can reach.

Chapter 8

Conclusion and Future Work

In this work we have scratched the surface of scaling applications to exploit available parallelism and utilize CGRA resources. We find that attaining high performance requires that the program formulation, the mapping and scheduling algorithms, and the CGRA accelerator architecture all align favorably. Due to the high complexity in the mapping and scheduling problem, we are not able to gain efficiency when we try to express more parallelism than the original REVEL workloads. We have shown empirically that increasing the CGRA size alone does not contribute to execution scalability, neither does introducing dynamic dataflow in isolation. Software-hardware cooperation is key to exploit the expressed parallelism, although we are not able to quantitatively show this, even when we use the current state-of-the-art compiler and simulator framework.

This thesis has raised more questions than it has answered, opening a vast amount of research opportunities. We find the following avenues of further work particularly interesting:

- Currently the REVEL architecture acts as a bimodal architecture that either computes spatially distributed programs at high performance and efficiency in the dedicated region, or utilize a less efficient temporal region to exploit other forms of less efficient computations. An opportunity here is to impose a static schedule onto the temporal region to fully exploit the temporal parallelism we can extract from pipelining computation. This would eliminate all serialization of performance-critical code we have encountered in this work, albeit at the cost of area.
- Reduce power dissipation impact on remapping to the temporal region. We can degrade the temporal PE by clock gating certain issue logic and possibly register files to make the system more efficient.
- Update scheduler heuristics so that we can express more parallelism in the application source code. This will require the ability to take into account that unrolling can be wider than a vector port. Distributing memory streams to multiple vector ports seems to be a key improvement to expose more parallelism. Without this ability the programmer has to manually partition the program into multiple similar loops, then annotate all of them as unique

DFGs. This is too much effort and the refactored code is harder to reason about.

- When we encounter triangular access patterns due to induction-variable dependencies, how can we balance the width of the vector interface to the set of problems we target? This requires empirical studies, and maybe profiling to identify appropriate tradeoff points.
- The previous point can be pivoted into modifying the vectorized computation at runtime by analyzing access patterns. This is interesting both from a static and dynamic view, as we can produce static schedules that can be improved by inspecting a small part of the execution state: the inductive variables.
- Automatic loop parallelization. Currently we have to express all parallelism in the program and rely on the compiler to build a DFG. Ideally, we would like for the compiler explore opportunities in scaling the DFG to exploit the parallelism without us explicitly setting the best-suited unrolling factor. To begin with, a hint of how much of the DFG we can unroll would be helpful, later we would like to annotate a region of code with e.g. `#pragma ss dedicated explode` to automatically fill available dedicated region resources with our critical path.
- Restrict search space of mapper and scheduler, at least when the DFG has multiple vector memory streams. This can enable more programs to map, either faster, or at all. In tandem with new heuristics this can increase usability of this architecture.

Bibliography

[1] K. Rupp, *48 Years of Microprocessor Trend Data*, original-date: 2018-02-15T05:10:17Z, 2020. [Online]. Available: `https://github.com/karlrupp/microprocessor-trend-data`.

[2] R. Courtland, 'The end of the shrink,' *IEEE Spectrum*, 2013.

[3] R. Dennard, F. Gaensslen, H.-N. Yu *et al.*, 'Design of ion-implanted MOSFET's with very small physical dimensions,' *IEEE Journal of Solid-State Circuits*, 1974.

[4] M. Anis and M. Elmasry, *Multi-Threshold CMOS Digital Circuits*. Springer US, 2003.

[5] H. Esmaeilzadeh, E. Blem, R. S. Amant *et al.*, 'Dark Silicon and the End of Multicore Scaling,'

[6] M. B. Taylor, 'Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse,' in *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, ACM Press, 2012.

[7] H. Sutter and J. Larus, 'Software and the concurrency revolution,' *Queue*, 2005.

[8] R. Hameed, W. Qadeer, M. Wachs *et al.*, 'Understanding sources of inefficiency in general-purpose chips,' *ACM SIGARCH Computer Architecture News*, 2010.

[9] S. Mittal and J. S. Vetter, 'A Survey of Methods for Analyzing and Improving GPU Energy Efficiency,' *ACM Computing Surveys*, 2014.

[10] L. A. Eggen, 'Towards Exploring Area-Performance Trade-Offs in Stream-Dataflow Architecture,'

[11] T. Nowatzki, V. Gangadhar, N. Ardalani *et al.*, 'Stream-Dataflow Acceleration,' in *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17*, ACM Press, 2017.

[12] J. Weng, S. Liu, Z. Wang *et al.*, 'A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms,'

[13] *PolyArch/dsa-apps*. [Online]. Available: `https://github.com/PolyArch/dsa-apps`.

[14] M. D. Hill and M. R. Marty, 'Amdahl's Law in the Multicore Era,'

[15] T. Nowatzki, V. Gangadhan, K. Sankaralingam *et al.*, 'Pushing the limits of accelerator efficiency while retaining programmability,' in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, ISSN: 2378-203X, 2016.

[16] *Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?: Queue: Vol 6, No 2*.

[17] L. Dagum and R. Menon, 'OpenMP: An industry standard API for shared-memory programming,' *IEEE Computational Science and Engineering*, 1998, Conference Name: IEEE Computational Science and Engineering.

[18] D. Chen and D. Singh, 'Invited paper: Using OpenCL to evaluate the efficiency of CPUS, GPUS and FPGAS for information filtering,' in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, ISSN: 1946-1488, 2012.

[19] D. Steinkraus, I. Buck and P. Simard, 'Using GPUs for machine learning algorithms,' in *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, ISSN: 2379-2140, 2005.

[20] A. Krizhevsky, I. Sutskever and G. E. Hinton, 'ImageNet classification with deep convolutional neural networks,' *Communications of the ACM*, 2017.

[21] S. Morishima and H. Matsutani, 'Accelerating Blockchain Search of Full Nodes Using GPUs,' in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, ISSN: 2377-5750, 2018.

[22] P. Xiang, Y. Yang and H. Zhou, 'Warp-level divergence in GPUs: Characterization, impact, and mitigation,' in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, ISSN: 2378-203X, 2014.

[23] B. Dally, 'The Future of Computing: Domain-Specific Accelerators,' [Online]. Available: `https://www.microarch.org/micro52/media/dally_keynote.pdf`.

[24] T. Nowatzki, V. Gangadhar, K. Sankaralingam *et al.*, 'Domain Specialization Is Generally Unnecessary for Accelerators,' *IEEE Micro*, 2017.

[25] M. D. Hill and V. J. Reddi, 'Accelerator-level Parallelism,' *arXiv:1907.02064 [cs]*, 2019, arXiv: 1907.02064.

[26] N. P. Jouppi, A. Borchers, R. Boyle *et al.*, 'In-Datacenter Performance Analysis of a Tensor Processing Unit,' in *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17*, ACM Press, 2017.

[27] N. P. Jouppi, D. H. Yoon, G. Kurian *et al.*, 'A domain-specific supercomputer for training deep neural networks,' *Communications of the ACM*, 2020.

[28] H. T. Kung and C. E. Leiserson, 'Systolic Arrays for (VLSI).,' CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1978, Section: Technical Reports.

[29] S. Chakradhar, M. Sankaradas, V. Jakkula *et al.*, 'A dynamically configurable coprocessor for convolutional neural networks,'

[30] D. Liu, T. Chen, S. Liu *et al.*, 'PuDianNao: A Polyvalent Machine Learning Accelerator,' *ACM SIGPLAN Notices*, 2015.

[31] L. Wu, A. Lottarini, T. K. Paine *et al.*, 'Q100: The architecture and design of a database processing unit,' *ACM SIGPLAN Notices*, 2014.

[32] O. Kocberber, B. Grot, J. Picorel *et al.*, 'Meet the walkers accelerating index traversals for in-memory databases,' in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[33] T. J. Ham, L. Wu, N. Sundaram *et al.*, 'Graphicionado: A high-performance and energy-efficient accelerator for graph analytics,' in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016.

[34] S. Rahman, N. Abu-Ghazaleh and R. Gupta, 'GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing,'

[35] G. Estrin, 'Organization of computer systems: The fixed plus variable structure computer,' in *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference on - IRE-AIEE-ACM '60 (Western)*, ACM Press, 1960.

[36] D. Bacon, R. Rabbah and S. Shukla, 'FPGA Programming for the Masses: The programmability of FPGAs must improve if they are to be part of mainstream computing.,' *Queue*, 2013.

[37] A. Canis, J. Choi, M. Aldham *et al.*, 'LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems,' *ACM Transactions on Embedded Computing Systems (TECS)*, 2013.

[38] *Vivado High-Level Synthesis*. [Online]. Available: `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[39] *Moonwalk | Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[40] 'Vivado Design Suite User Guide: Partial Reconfiguration (UG909),' 2018.

[41] R. Ferreira, V. Duarte, W. Meireles *et al.*, 'A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures,' in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, IEEE, 2013.

[42] *AWS Inferentia - Amazon Web Services (AWS)*. [Online]. Available: `https://aws.amazon.com/machine-learning/inferentia/`.

[43] M. Annaratone, E. Arnould, T. Gross *et al.*, 'The Warp Computer: Architecture, Implementation, and Performance,' *IEEE Transactions on Computers*, 1987.

[44] J. B. Dennis and D. P. Misunas, 'A preliminary architecture for a basic dataflow processor,' in *Proceedings of the 2nd annual symposium on Computer architecture*, ser. ISCA '75, Association for Computing Machinery, 1974.

[45] Dennis, 'Data Flow Supercomputers,' *Computer*, 1980, Conference Name: Computer.

[46] T. Nowatzki, N. Ardalani, K. Sankaralingam *et al.*, 'Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign,' in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques - PACT '18*, ACM Press, 2018.

[47] J. R. Gurd, C. C. Kirkham and I. Watson, 'The Manchester prototype dataflow computer,' *Communications of the ACM*, 1985.

[48] Arvind and R. Nikhil, 'Executing a program on the MIT tagged-token dataflow architecture,' *IEEE Transactions on Computers*, 1990, Conference Name: IEEE Transactions on Computers.

[49] Y. Liu, X. Zhao, M. Jahre *et al.*, 'Get Out of the Valley: Power-Efficient Address Mapping for GPUs,' in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018.

[50] K. Asanović and D. A. Patterson, 'Instruction Sets Should Be Free: The Case For RISC-V,' EECS Department, University of California, Berkeley, Tech. Rep., 2014.

[51] N. Binkert, B. Beckmann, G. Black *et al.*, 'The gem5 simulator,' *ACM SIGARCH Computer Architecture News*, 2011.

Appendix A

Modified GEMM_M

Code listing A.1: Modified gemm (MachSuite)

```
void bbgemm(TYPE m1[N], TYPE m2[N], TYPE prod[N])
{
  #pragma ss config
  {
    int i, k, j, jj, kk;
    int i_row, k_row;
    TYPE temp_x, mul;

    for (jj = 0; jj < row_size; jj += block_size) {
      for (kk = 0; kk < row_size; kk += block_size) {
        for (i = 0; i < row_size; ++i) {

          #pragma ss stream nonblock
          for (k = 0; k < block_size; ++k) {
            i_row = i * row_size;
            k_row = (k + kk) * row_size;
            temp_x = m1[i_row + k + kk];

            #pragma ss dfg dedicated unroll(U)
            for (j = 0; j < block_size; ++j) {
              mul = temp_x * m2[k_row + j + jj];
              prod[i_row + j + jj] += mul;
            }
          }
        }
      }
    }
  }
}
```