SCALING A RECONFIGURABLE DATAFLOW ACCELERATOR

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Yaqi Zhang
May 2020

Abstract

With the slowdown of Moore's Law, specialized hardware accelerators are gaining tractions as energy-efficient platforms that deliver 100-1000x performance improvement over general-purpose processors. Reconfigurable architectures are particularly promising in providing high-throughput and low-latency computation in streaming and data-intensive analytics applications. Instead of dynamically executing instructions like in processor architectures, reconfigurable architectures have flexible datapath that can be statically configured to parallelize and/or pipeline the program spatially across on-chip resources. The pipelined execution model and explicitly-managed scratchpad in reconfigurable accelerators dramatically reduce the performance, area, and energy overhead in dynamic execution and conventional cache memory hierarchy, respectively. Plasticine is a hierarchical coarse-grained reconfigurable dataflow accelerator introduced at Stanford in 2017. Compared to fine-grained reconfigurable architecture, like FPGAs, Plasticine has shown 76x performance/watt benefit due to the reduction in routing overhead and the improvement in on-chip resource density.

In this talk, we will discuss two aspects of the software-hardware codesign that impact the accessibility and performance of the accelerator. One of the biggest challenges that hinders the adoption of these accelerators is the low-level declarative configuration interface that requires the programmers to have detailed knowledge about the underlying microarchitecture implementation and hardware constraints. To address the challenge, I will introduce a compiler stack that provides a high-level programming interface that efficiently translates imperative control constructs to streaming dataflow execution with minimum

synchronization overhead on an on-chip distributed architecture. The compiler handles the hardware constraints systematically with resource virtualization. Next, I will present a comprehensive study on the on-chip network design for reconfigurable dataflow architectures that sustain performance in a scalable fashion with high energy efficiency.

Acknowledgements

I would like to thank my mother and the little green men from Mars.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the end of Dennard Scaling [**?**], the amount of performance one can extract from a CPU is reaching a limit. To provide general-purpose flexibility, CPU spends the majority of energy on overheads, including dynamic-instruction execution, branch prediction, and a cache hierarchy, and less than 20% of the energy on the actual computation [**?**]. Even worse, the power wall is limiting the entire multicore family to reach the doubled performance improvement per generation enabled by technology scaling in the past[**?**].

Chapter 2

Background

2.1 Execution Schedule of Spatial Architectures

The biggest advantage of reconfigurable accelerators, compared to processor-based architectures such as CPUs and GPUs, is the ability to explore pipeline parallelism at multiple granularity.

In traditional Von Neumann architectures, a computer consists of a processing unit that performs computation, a memory unit that stores the program states, and a control unit that tracks execution states and fetch the instruction to execute. This computing model inheritely assumes that instructions with in a program are executed in time.

Unlike traditional Von Neumann architectures, which inheritely assumes program is executed in time, reconfigurable data-flow architecutes can statically program pipelined access reduce off-chip access. HBM small capacity scratchpad improves effective bandwidth and capacity

Figure 2.1 shows an example program executed on the

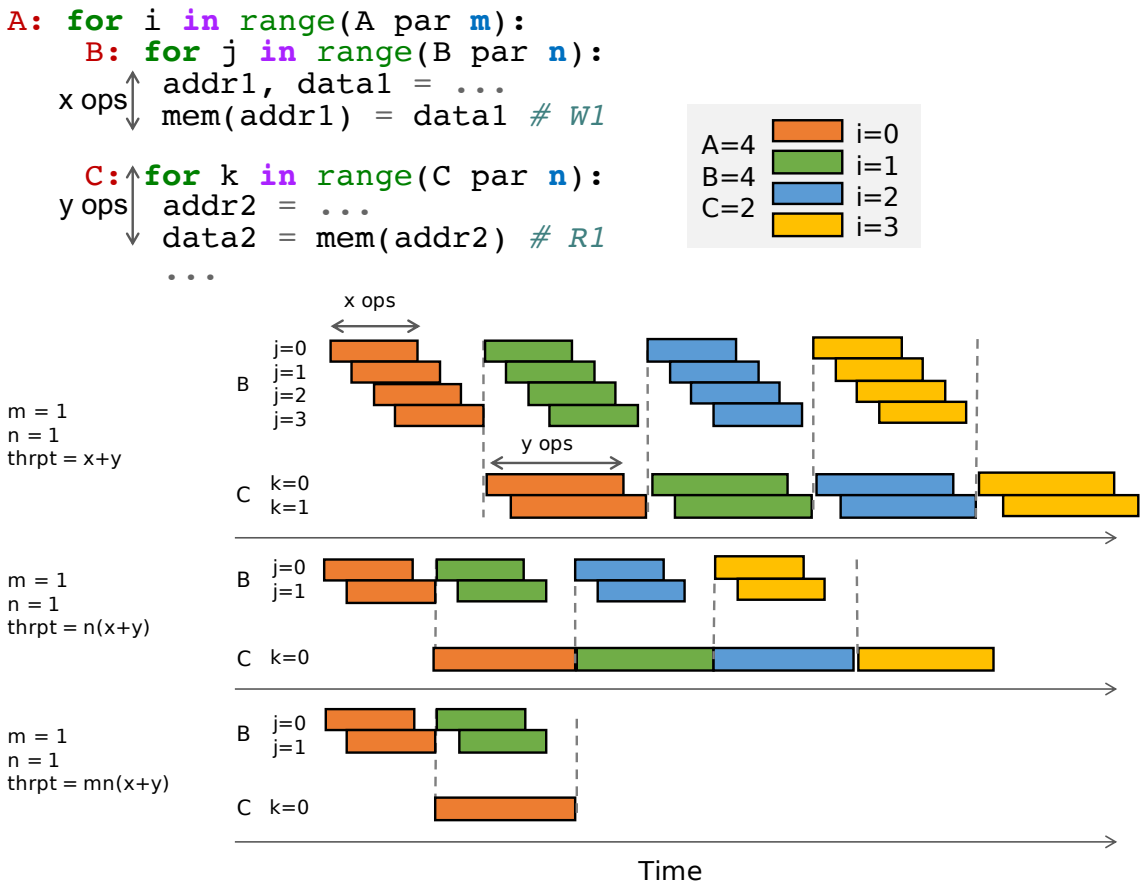


Figure 2.1: Hiearchical pipelining and parallelization in spatial architecture. (a) illustrates the runtime and throughput of a hiearchically pipelined and parallelized program on a reconfigurable spatial architecture. At inner level, instructions within each basic block are fine-grained pipelined across iterations of the inner most loop. At outer level, the inner loops are coarse-grained pipeliend across the outer loop iterations. Exploting multiple levels of pipeline parallelism gives a total throughput of $x + y$ operations per cycle. (b) Vectorizing the inner most loops B and C by n increases the throughput to $(x + y)n$. (c) Parallelizing the outer loop A by m furhter increases the throughput to $(x + y)mn$.

Concurrency Level	Instruction	Data	Task/Kernel
Parallelsim	CPU,RDA	CPU,GPU,RDA	CPU,RDA
Pipelining	RDA	RDA	RDA

Table 2.1: Concurrency level explored by different architectures

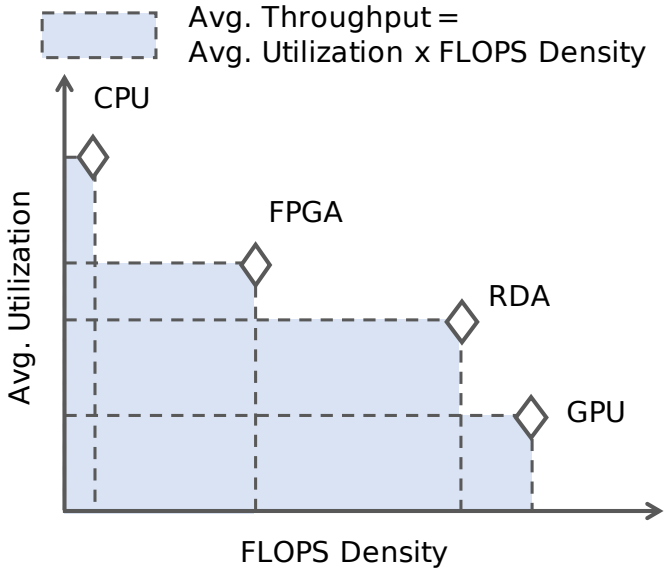


Figure 2.2: Average utilization vs. peak compute density tradeoff among different architectures.

$$\text{thrpt}_{\text{app}} = \min\left(\begin{array}{c}\text{thrpt}_{\text{comp}},\\ \frac{\text{comp}}{\text{access}_{\text{off}}}\text{BW}_{\text{off}},\\ \frac{\text{comp}}{\text{access}_{\text{on}}}\text{BW}_{\text{on}},\\ \frac{\text{comp}}{\text{trans}_{\text{net}}}\text{BW}_{\text{net}}\end{array}\right)$$

Throughput	Proportional To
Compute	P, D
Off-chip Memory	P, D
On-chip Memory	P
On-chip Network	P^{-}, D

Application-specific
Hardware-specific
P: Parallelization factor
D: Pipelining depth

Figure 2.3: High-level performance model of spatial architectures

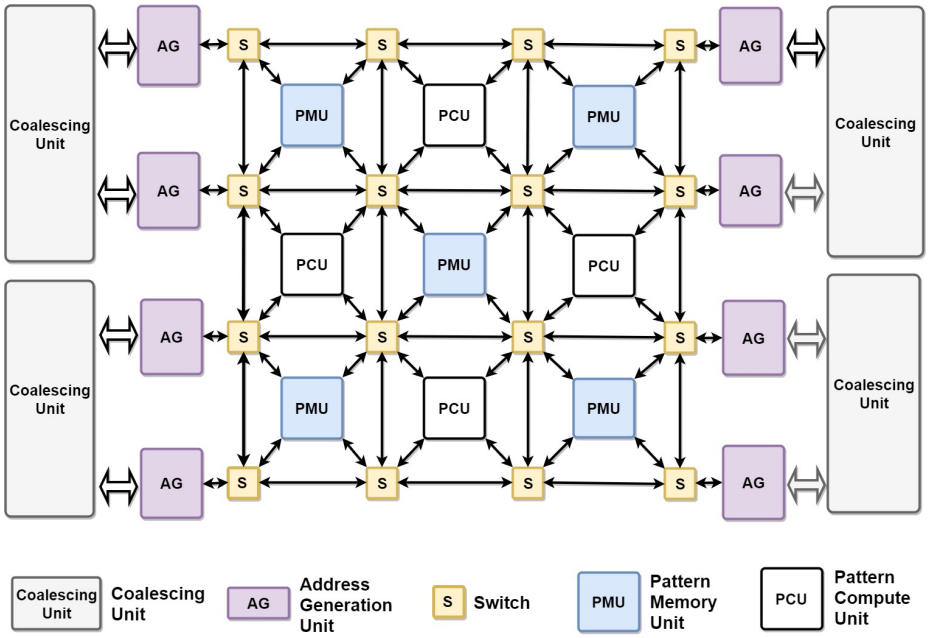


Figure 2.4: Plasticine chip-level architectural diagram

```
// Host to accelerator register for scalar input with
// user annotated value
val N = ArgIn[Int]; bound(N) = 1024
// 1-D DRAM size in N
val vecA, vecB = DRAM[T](N)
// 2-D DRAM size in NxN
val matC = DRAM[T](N, N)
// Loop unrolling factors
val op1, op2, ip:Int = ...
// Blocking sizes of vecA and vecB
val tsA, tsB:Int = ...

// Accelerator kernel
C0: Accel {
  // C1 is parallelized by op1
  C1: Foreach(min=0, step=tsA, max=N, par=op1){ i =>
    // Allocate 1-D scratchpad size in tsA
    val tileA = SRAM[T](tsA)
    // Load range i to i+tsA of vectorA from off- to
    // on-chip parallelized by ip
    C2: tileA load vecA(i::i+tsA par ip)
    C3: Foreach(min=0, step=tsB, max=N, par=op2) { j =>
      val tileB = SRAM[T](tsB)
      C4: tileB load vecB(j::j+tsB par ip)
      // 2-D scratchpad
      val tileC = SRAM[T](tsA, tsB)
      C5: Foreach(min=0, step=1, max=tsA){ ii =>
        Foreach(min=0, step=1, max=tsB, par=ip) { jj =>
          tileC(ii, jj) = tileA(ii) * tileB(jj)
        }
      }
      // Store partial results to DRAM
      C6: matC(i::i+tsA, j::j+tsB par ip) store tileC
    }
  }
}
```

Figure 2.5: Example of outer product in Spatial pseudocode.

2.2 Plasticine

2.3 Spatial

We use Spatial, an open source domain specific language for reconfigurable accelerators, to target spatial architectures [3]. Spatial describes applications with nested loops and an explicit memory hierarchy that captures data movement on-chip and off-chip. This exposes design parameters that are essential for achieving high performance on spatial architectures, including blocking size, loop unrolling factors, inner-loop pipelining, and coarse-grained pipelining of arbitrarily nested loops. To enable loop-level parallelization and pipelining, Spatial automatically banks and buffers intermediate memories between loops. An example of outer product—element-wise multiplication of two vectors resulting in a matrix—in Spatial is shown in Figure 2.5. For spatial architectures, Design Space Exploration (DSE) of parameters (e.g., *op1*, *op2*, *ip*, *tsA*, *tsB*) is critical to achieve good resource utilization and performance [4].

Chapter 3

Architecture

In this section, we discuss the architectural advancement on top of the original Plasticine architecture introduced in [5]. These architectural additions helps increase the application coverage or improve the mapping strategies of existing applications by supporting new language constructs, data types, and improves the utilizations of the hardware. Specifically, Section 3.1 lay outs the datapath changes in order to support more flexible banking schemes required by general access patterns supported in Spatial; Section 3.2 discusses the hardware specialization and architectural sizing for machine learning applications; **??** provides an extensive study on on-chip network selection reconfigurable spatial architectures.

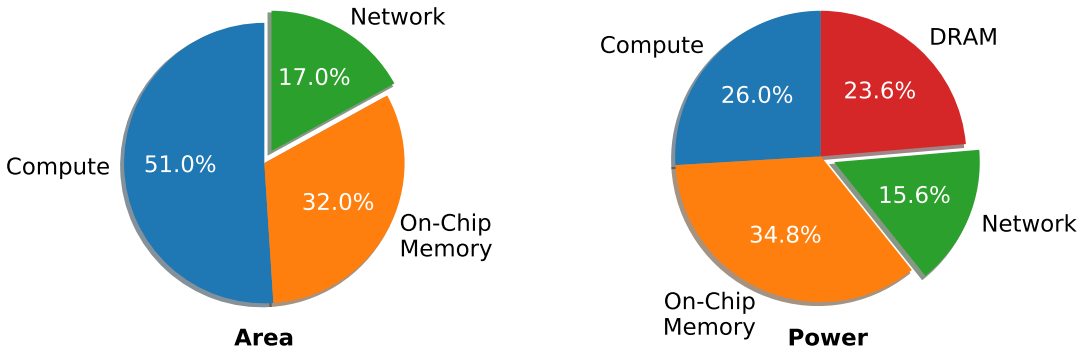


Figure 3.1: Area and power breakdown of Plasticine

3.1 Generic Banknig Support

3.2 Plasticine Specialization for RNN Serving

To show efficient execution of the loop and parallel pattern constructs, we map our implementation onto a spatial architecture, Plasticine. **Foreach** at Line $17, 19$ and **Reduce** at Line $22, 23$ are mapped to PCUs on Plasticine. When the application size is small, these constructs are executed using pipelined SIMD lanes within a single PCU. When the application size is large, multiple PCUs can be used to parallelize and pipeline the dot product across PCUs. Element-wise operations can be executed in a deep pipeline formed by chaining multiple PCUs.

To fit an RNN's weights on-chip, we execute our application with low-precision arithmetics. In this section, we propose the necessary micro-architectural changes to support low-precision arithmetics on Plasticine. We also discuss architectural parameter selection for Plasticine to serve RNN applications efficiently.

3.2.1 Mixed-Precision Support

Previous works [6, 7] have shown that low-precision inference can deliver promising performance improvements without sacrificing accuracy. In the context of reconfigurable architectures such as FPGAs, low-precision inference not only increases compute density, but also reduces required on-chip capacity for storing weights and intermediate data.

To support low-precision arithmetics without sacrificing coarse-grained reconfigurability, we introduce two low-precision struct types in Spatial: a tuple of 4 8-bit and 2 16-bit floating-point numbers, `4-float8` and `2-float16` respectively. Both types packs multiple low-precision values into a single precision storage. We support only 8 and 16-bit precisions, which are commonly seen in deep learning inference hardwares. Users can only access values that are 32-bit aligned. This constraint guarantees that the microarchitectual change is only local to the PCU. Banking and DRAM access granularity remains intact from the original design.

Figure 3.2 (a) shows the original SIMD pipeline in a Plasticine PCU. Each FU supports both floating-point and fix-point operations. When mapping applications on Plasticine,

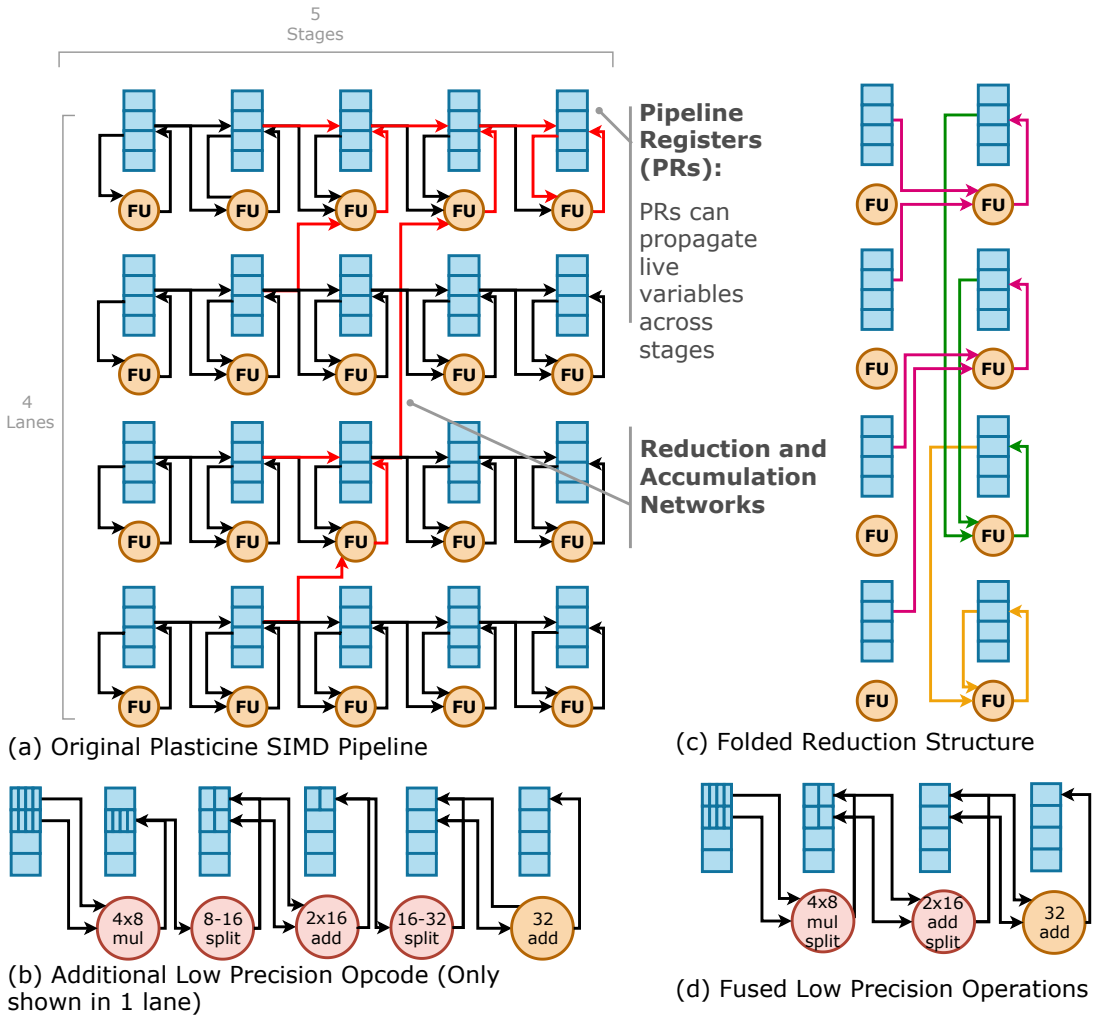


Figure 3.2: Plasticine PCU SIMD pipeline and low-precision support. Red circles are the new operations. Yellow circles are the original opertaions in Plasticine. In (d) the first stage is fused $1^{st}, 2^{nd}$ stages, and the second stage is fused 3^{nd}, 4^{th} stages of (b).

the inner most loop body is vectorized across the lanes of the SIMD pipeline, and different operations of the loop body are mapped to different stages. Each pipeline stage contains a few pipeline registers (PRs) that allow propagation of live variables across stages. Special cross-lane connections as shown in red in Figure 3.2 enable reduction operations. To support 8-bit element-wise multiplication and 16-bit reduction, we add 4 opcodes to the FU, shown in Figure 3.2 (b). The 1^{st} and 3^{rd} stages are element-wise, low-precision operations that multiply and add 4 8-bit and 2 16-bit values, respectively. The 2^{nd} and 4^{th} stages rearrange low-precision values into two registers, and then pad them to higher precisions. The 5^{th} stage reduces the two 32-bit value to a single 32-bit value using the existing add operation. From here, we can use the original reduction network shown in Figure 3.2 (a) to complete the remaining reduction and accumulates in 32-bit connection.

With 4 lanes and 5 stages, a PCU first reads 16 8-bit values, performs 8-bit multiplication followed by rearrangement and padding, and then produce 16 16-bit values after the second stage. The intermediate values are stored in 2 PRs per lane. Next, 16 16-bit values are reduced to 8 16-bit values and then rearranged to 8 32-bit value in 2 PRs per lane. Then, the element-wise addition in 32-bit value reduces the two registers in each line into 4 32-bit values. These values are fed through the reduction network that completes the remaining reduction and accumulation in two plus one stages.

In a more aggressive specialization, we can fuse the multiply and rearange into the same stage. We also fuse the first low-precision reduction with the next rearange as shown in Figure 3.2 (d). In this way, we can perform the entire low-precision map-reduce in 2 stages in addition to the original full precision reduction. In order to maximize hardware reuse, we assume that it is possible to construct a full precision FU using low-precision FUs. In addition, we observe that the original reduction network in the SIMD lanes could lead to low FU utilization. To improve FU utilization, we fold the entire tree structure in a single stage. Figure 3.2 (c) shows the folded reduction accumulation structure. Specifically, latter reductions in the tree are mapped to earlier stages in the pipeline. In this setup, the entire reduction plus accumulation is still fully pipelined in $\log_2(\#_{LANE})+1$ cycles with no structural hazard. With fused reduced-precision multiplication and reduction, and folded

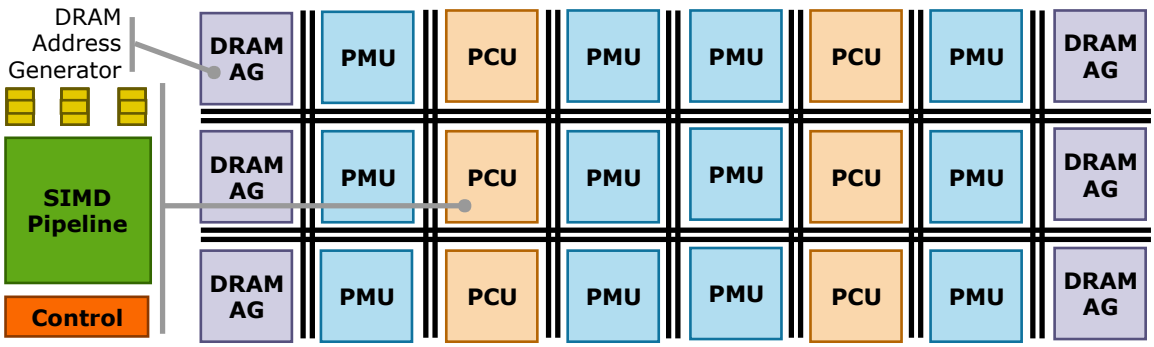


Figure 3.3: Variant configuration of Plasticine for serving RNN.

reduction tree, a PCU is able to perform all map-reduce that accumulates $4\#_{LANE}$ 8-bit values using 4 stages. All the operations are completed in $2 + \log_2(\#_{LANE}) + 1$ cycles.

3.2.2 Sizing Plasticine for RNN Serving

Evaluating an RNN cell containing N hidden units and N input features requires $2N^2$ computations and $N^2 + N$ memory reads. With large N, the compute to memory ratio is 2:1. The original Plasticine architecture uses a checkerboard layout with 1 to 1 ratio between PCU and PMU. A PCU has 6 stages and 16 lanes, and a PMU has 16 banks. This provides a 6:1 ratio between compute resource and on-chip memory read bandwidth. As a result of this layout, on-chip memory read bandwidth becomes the bottleneck for accelerating RNN serving applications. Given that RNNs cover a wide range of important applications, we select a Plasticine configuration tailored for RNN serving. Specifically, we choose a 2 to 1 PMU-PCU ratio with 4 stages in each PCU. Figure 3.3 shows the layout of this Plasticine variant.

3.3 On-chip Network

This section discusses communication characteristics common in applications that have been spatially mapped to CGRAs. Because CGRAs encompass a broad range of architectures, we first describe the abstract machine model of our target CGRA for this study, shown in Figure 3.3. The CGRA contains Physical Blocks (PBs) corresponding to distributed hardware resources, including compute units, scratchpads, and DRAM controllers. The communication between PBs, sent over a reconfigurable network, is purely streaming. Compute PBs have a simple control mechanism: they wait on input data dependencies and stall for backpressure from the network. The network guarantees exactly-once, in-order delivery with variable latency, and communication between PBs can have varying granularities (e.g., 512-bit vector or 32-bit scalar).

In this study, we focus on two categories of CGRA architectures. The first architecture uses pipelining in compute PBs, as shown in Figure 3.3. To provide high throughput, each stage of the pipeline exploits SIMD parallelism, and multiple SIMD operations are pipelined within a PB. Plasticine, a recently proposed CGRA, is an example of a pipelined architecture [5].

The second architecture uses time-scheduled execution, where each PB executes a small loop of instructions (e.g., 6) repeatedly. The scheduling window is small enough that instructions are stored as part of the configuration fabric, without dynamic instruction fetch overhead. This execution model creates more interleaved pipelining across PBs with communication that is tolerant of lower network throughput, which provides an opportunity to share links. Many proposed CGRAs and domain-specific architectures use this *time-scheduled* form of computation, including Brainwave [**?**] and DaDianNao [**?**].

3.3.1 Application Characteristics

The requirements of an interconnection network are a function of the communication pattern of the application, underlying CGRA architecture, and compilation process. We identify the following key characteristics of spatially mapped applications:

Vectorized communication

Recent hardware accelerators use large-granularity compute tiles (e.g., vectorized compute units and SIMD pipelines) for SIMD parallelism [5, **?**], which improves compute density while minimizing control and configuration overhead. Coarser-grained computation typically increases the size of communication, but glue logic, reductions, and loops with carried dependencies (i.e., non-parallelizable loops) contribute to scalar communications. This variation in communication motivates specialization for optimal area- and energy-efficiency: separate networks for different communication granularities.

Broadcast and incast communication

A key optimization for spatial reconfigurable accelerators is the parallelization of execution across PBs. This parallelization involves unrolling outer loop nests in addition to the vectorization of the inner loop. For neural network accelerators, this corresponds to parallelizing one layer across different channels. By default, pipeline parallelism involves one-to-one communication between dependent stages. However, when a consumer stage is parallelized, the producer sends a one-to-many broadcast to all of its consumers. Similarly, when a producer stage is parallelized, all partial results are sent to the consumer, forming a many-to-one incast link. When both the producer and the consumer are parallelized, the worst case is many-to-many communication, because the parallelized producers may dynamically alternate between parallelized receivers.

Compute to memory communication

To encourage better sharing of on-chip memory capacity, many accelerators have shared scratchpads, either distributed throughout the chip or on its periphery [5, **?**, **?**]. Because the compute unit has no local memory to buffer temporary results, the results of all computations are sent to memory through the network. This differs from the NoCs used in multi-processors, where each core has a local cache to buffer intermediate results. Studies have shown that for large-scale multi-processor systems, network latency—not throughput—is

the primary performance limiter [**?**]. For spatial accelerators, however, compute performance is limited by network throughput, and latency is comparatively less important.

Communication-aware compilation

Unlike the dynamic communication of multi-processors, communication on spatial architectures is created statically by compiling and mapping the compute graph onto the distributed PB resources. As the compiler performs optimization passes, such as unrolling and banking, it has static knowledge about communication generated by these transformations. This knowledge allows the compiler to accurately determine which network flows in the transformed design correspond to throughput-critical inner-loop traffic and which correspond to low-bandwidth outer-loop traffic.

We select a mix of applications from domains where hardware accelerators have shown promising performance and energy-efficiency benefits, such as linear algebra, databases, and machine learning. Table 3.1 lists the applications and their data size. Figure 3.4 shows, for each design, which resource limits performance: compute, on-chip memory, or DRAM bandwidth. DotProduct, TPCHQ6, OuterProduct, and BlackScholes are DRAM bandwidth-bound applications. These applications use few on-chip resources to achieve maximum performance, resulting in minimal communication. Lattice (a fast inference model for low-dimensional regression [**?**]), GDA, Kmeans, SGD, and LogReg are compute-intensive applications; for these, maximum performance requires using as much parallelization as possible. Finally, LSTM, GRU, and LeNet are applications that are limited by on-chip memory bandwidth or capacity. For compute- and memory-intensive applications, high utilization translates to a large interconnection network bandwidth requirement to sustain application throughput.

Figure 3.5(a,b) shows the communication pattern of applications characterized on the pipelined CGRA architecture, including the variation in communication granularity. Compute and on-chip memory-bound applications show a significant amount of high-bandwidth communication (links with almost 100% activity). A few of these high-bandwidth links

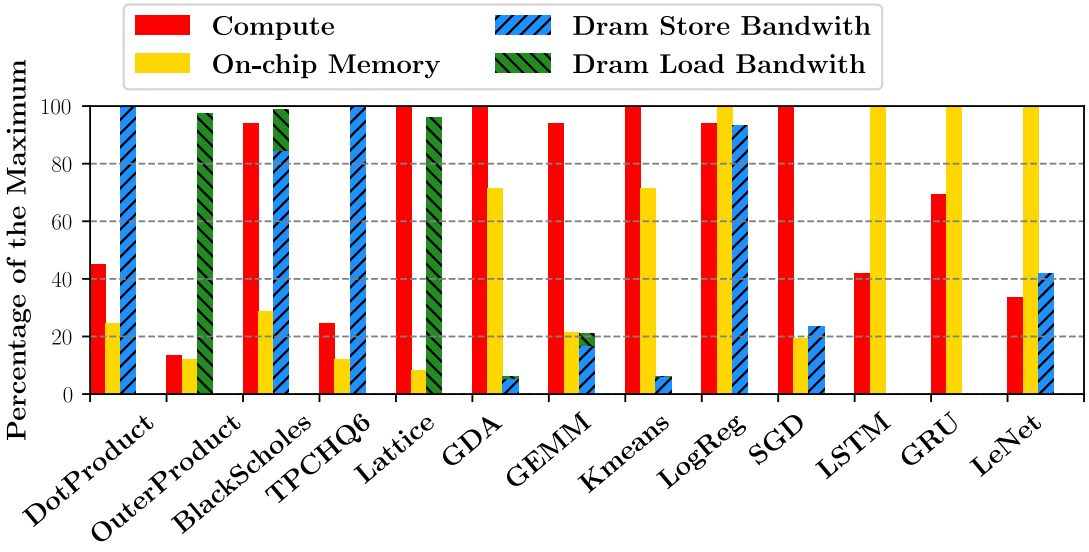
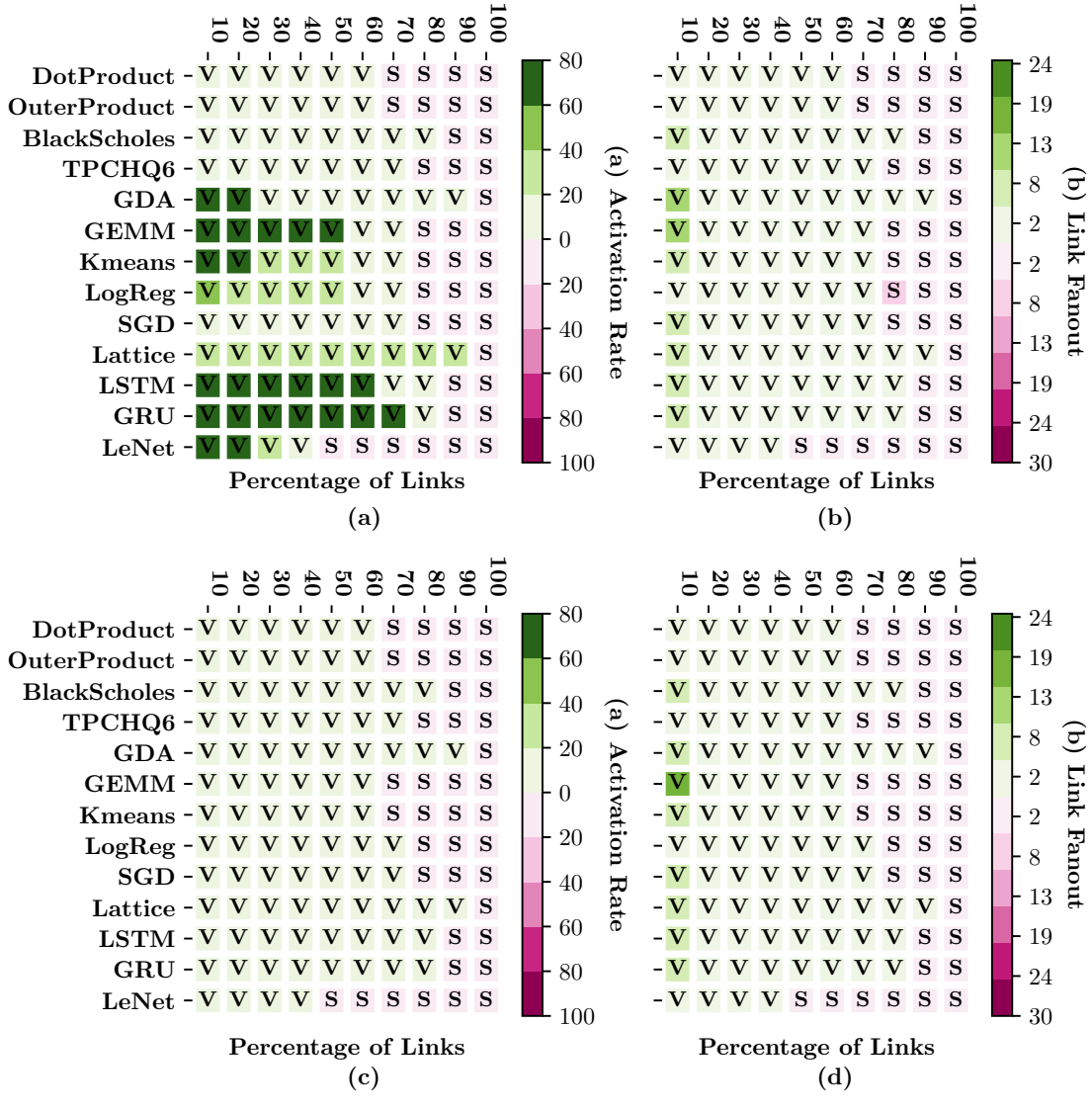


Figure 3.4: Physical resource and bandwidth utilization for various applications.



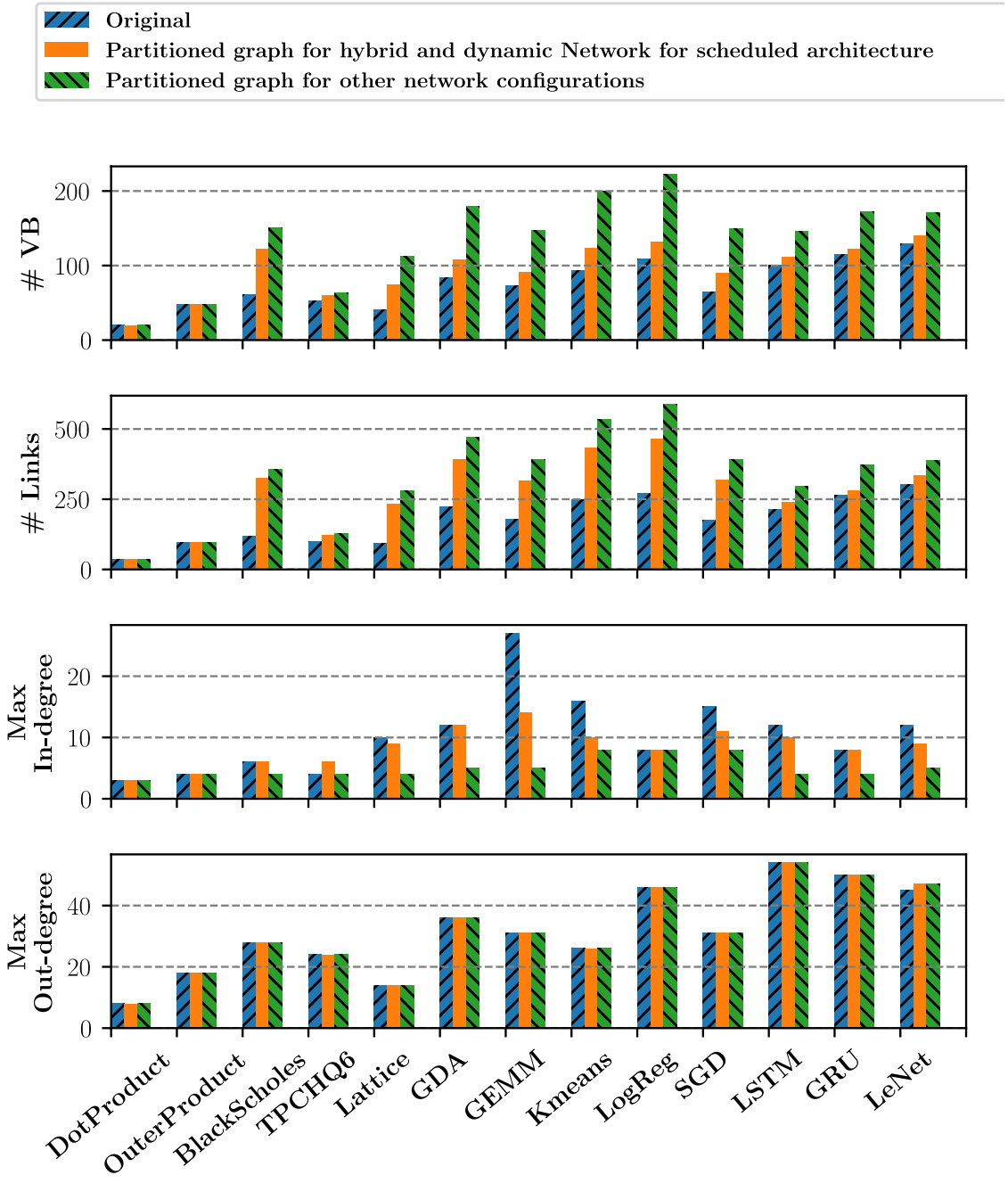


Figure 3.6: Characteristics of program graphs.

also exhibit high broadcast fanout. Therefore, a network architecture must provide sufficient bandwidth and efficient broadcasts to sustain program throughput. On the contrary, time-scheduled architectures, shown in Figure 3.5(c,d), exhibit lower bandwidth requirements due to the lower throughput of individual compute PBs. Even applications limited by on-chip resources have less than a 30% firing rate on the busiest logical links; this reveals an opportunity for link sharing without sacrificing performance.

Figure 3.6 shows statistics describing the VB dataflow graph before and after partitioning. The blue bars show the number of VBs, number of logical links, and maximum VB input/output degrees in the original parallelized program; the yellow and green bars show the same statistics after partitioning. Fewer VBs are partitioned for hybrid networks and dynamic networks with the time-scheduled architecture, as explained in Section **??**. The output degree does not change with partitioning because most outputs with a large degree are from broadcast links.

3.3.2 Design Space for Network Architectures

We start with several statically allocated network designs, where each SIMD pipeline connects to several switches, and vary flow control strategies and network bisection bandwidth. In these designs, each switch output connects to exactly one switch input for the duration of the program. We then explore a dynamic network, which sends program data as packets through a NoC. The NoC uses a table-based routing scheme at each router to allow for arbitrary routes and tree-based broadcast routing. Finally, we explore the benefits of specialization by evaluating design points that combine several of these networks to leverage the best features of each.

Static networks

We explore static network design points along three axes. First, we study the impact of flow-control schemes in static switches. In credit-based flow control [**?**], the source and destination PBs coordinate to ensure that the destination buffer does not overflow. For this

design point, switches only have a single register at each input, and there is no backpressure between switches. The alternate design point uses a skid-buffered queue with two entries at each switch; using two entries enables per-hop backpressure and accounts for a one-cycle delay in stalling the upstream switch. At full throughput, the receiver will consume data as it is sent and no queue will ever fill up. The second axis studied is the bandwidth, and therefore routability, of the static network. We vary the number of connections between switches in each direction, which trades off area and energy for bandwidth. Finally, we explore specializing static links: using a separate scalar network to improve routability at a low cost.

Dynamic networks

Our primary alternate design is a dynamic NoC using per-hop virtual channel flow control. Routing and Virtual Channel (VC) assignment are table-based: the compiler performs static routing and VC allocation, and results are loaded as a part of the routers' configurations at runtime. The router has a separable, input-first VC and switch allocator with a single iteration and speculative switch allocation [**?**]. Input buffers are sized just large enough (3 entries) to avoid credit stalls at full throughput. Broadcasts are handled in the network with duplication occurring at the last router possible to minimize energy and congestion. To respect the switch allocator's constraints, each router sends broadcasts to output ports sequentially and in a fixed order. This is because the switch allocator can only grant one output port per input port in every cycle, and the RTL router's allocator does not have sufficient timing slack to add additional functionality. We also explore different flit widths on the dynamic network, with a smaller bus taking multiple cycles to transmit a packet.

Because CGRA networks are streaming—each PB pushes the result to the next PB(s) without explicit request—the network cannot handle routing schemes that may drop packets; otherwise, application data would be lost. Because packet ordering corresponds directly to control flow, it is also imperative that all packets arrive in the order they were sent; this further eliminates adaptive or oblivious routing from consideration. We limit our study of dynamic networks to statically placed and routed source routing due to these

architectural constraints. PBs propagate backpressure signals from their outputs to their inputs, so they must be considered as part of the network graph for deadlock purposes [**?**]. Furthermore, each PB has fixed-size input buffers; these are far too small to perform high-throughput, end-to-end credit-based flow control in the dynamic network for the entire program [**?**]. Practically, this means that no two logical paths may be allowed to conflict at *any* point in the network; to meet this guarantee, VC allocation is performed to ensure that all logical paths traversing the same physical link are placed into separate buffers.

Hybrid networks

Finally, we explore hybrids between static and dynamic networks that run each network in parallel. During static place and route, the highest-bandwidth logical links from the program graph are mapped onto the static network; once the static network is full, further links are mapped to the dynamic network. By using compiler knowledge to identify the relative importance of links—the link fanout and activation factor—hybrid networks can sustain the throughput requirement of most high-activation links while using the dynamic network for low-activation links.

3.3.3 Performance, Area, and Energy Modeling

We use a cycle-accurate simulator to model the pipeline and scheduling delay for the two types of architectures, integrated with DRAMSim [**?**] to model DRAM access latency. For static networks, we model a distance-based delay for both credit-based and per-hop flow control. For dynamic networks, we integrate our simulator with Booksim [**?**], adding support for arbitrary source routing using look-up tables. Finally, to support efficient multicasting in the dynamic network, we modify Booksim to duplicate broadcast packets at the router where their paths diverge. At the divergence point, the router sends the same flit to multiple output ports over multiple cycles. We assume each packet carries a unique ID that is used to look up the output port and next VC in a statically generated routing table, and that the ID is roughly the same size as an address. When the packet size is greater than the

Benchmark	Description	Data Size
DotProduct	Inner product	1048576
OuterProduct	Outer product	1024
BlackScholes	Option pricing	1048576
TPCHQ6	TPC-H query 6	1048576
Lattice	Lattice regression [**?**]	1048576
GDA	Gaussian discriminant analysis	127×1024
GEMM	General matrix multiply	$256 \times 256 \times 256$
Kmeans	K-means clustering	k=64, dim=64, n=8192, iter=2
LogReg	Logistic regression	8192×128, iter=4
SGD	Stochastic gradient descent for a single layer neural network	16384×64, epoch=10
LSTM	Long short term memory recurrent neural network	1 layer, 1024 hidden units, 10 time steps
GRU	Gated recurrent unit recurrent neural network	1 layer, 1024 hidden units, 10 time steps
LeNet	Convolutional neural network for character recognition	1 image

Table 3.1: Benchmark summary

flit size, the transmission of a single packet takes multiple cycles.

Area and power

To efficiently evaluate large networks, we start by characterizing the area and power consumption of individual routers and switches used in various network configurations. The total area and energy are then aggregated over all switches and routers in a particular network. We use router RTL from the Stanford open source NoC router [**?**] and our own parameterized switch implementation. We synthesize using Synopsys Design Compiler with a $28\,\mathrm{nm}$ technology library and clock-gating enabled, meeting timing at a 1 GHz clock frequency. Finally, we use Synopsys PrimeTime to back-annotate RTL signal activity to the post-synthesis switch and router designs to estimate gate-level power.

We found that power consumption can be broken into two types: inactive power consumed when switches and routers are at zero-load (P_{inactive}, which includes both dynamic

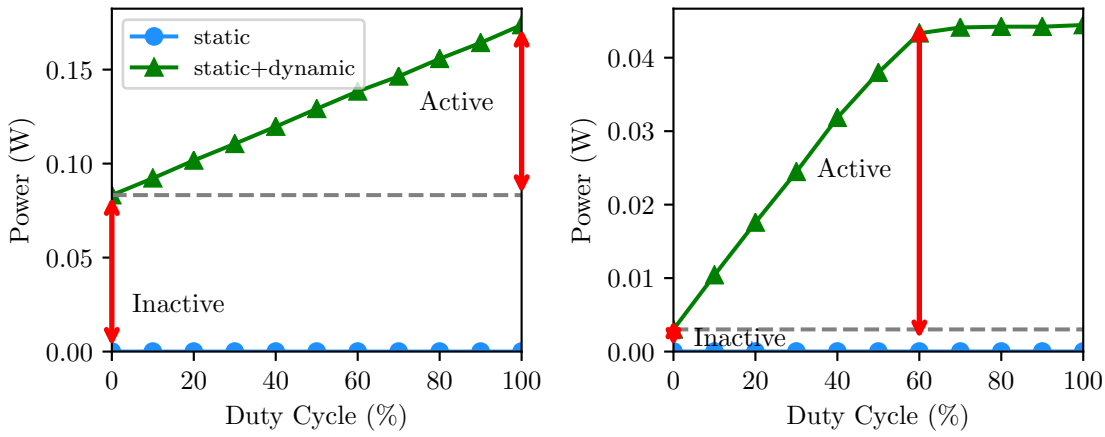


Figure 3.7: Switch and router power with varying duty cycle.

and static power), and active power. The active power, as shown in Section 3.3.3, is proportional to the amount of data transmitted. Because power scales linearly with the amount of data movement, we model the marginal energy to transmit a single flit of data (flit energy, E_{flit}) by dividing active energy by the number flits transmitted in the testbench:

$$E_{\text{flit}} = \frac{(P - P_{\text{inactive}})\, T_{\text{testbench}}}{\#\text{flit}} \tag{3.1}$$

While simulating an end-to-end application, we track the number of flits transmitted at each switch and router in the network, as well as the number of switches and routers allocated by place and route. We assume unallocated switches and routers are perfectly power-gated, and do not consume energy. The total network energy for an application on a given network (E_{net}) can be computed as:

$$E_{\text{net}} = \sum_{\text{allocated}} P_{\text{inactive}} T_{\text{sim}} + E_{\text{flit}} \#\text{flit}, \tag{3.2}$$

where P_{inactive}, E_{flit}, and $\#$flit are tabulated separately for each network resource.

Figure 3.10 shows that switch and router power scale linearly with the rate of data transmission, but that there is non-zero power at zero-load. For simulation, the duty cycle refers

to the amount of offered traffic, not accepted traffic. Because our router uses a crossbar without speedup [**?**], the testbench saturates the router at 60% duty cycle when providing uniform random traffic. Nonetheless, router power still scales linearly with accepted traffic.

A sweep of different switch and router parameters is shown in Figure 3.8. Subplots (d,e,f) show the energy necessary to transmit a single bit through a switch or router. Subplot (a) shows the roughly quadratic scaling of switch area with the number of links between adjacent switches. Vector switches scale worse with increasing bandwidth than scalar switches, mostly due to increased crossbar wire load. At the same granularity, a router consumes more energy a switch to transmit a single bit of data, even though the overall router consumes less power (as shown in Figure 3.10); this is because the switch has a higher throughput than the router. The vector router has lower per-bit energy relative to the scalar router because it can amortize the cost of allocation logic, whereas the vector switch has higher per-bit energy relative to the scalar switch due to increased capacitance in the large crossbar. Increasing the number of VCs or buffer depth per VC also significantly increases router area and energy, but reducing the router flit width can significantly reduce router area.

Overall, these results show that scaling static bandwidth is cheaper than scaling dynamic bandwidth, and a dynamic network with small routers can be used to improve link sharing for low bandwidth communication. We also see that a specialized scalar network, built with switches, adds negligible area compared to and is more energy efficient than the vector network. Therefore, we use a static scalar network with a bandwidth of 4 for the remainder of our evaluation, except when evaluating the pure dynamic network. The dynamic network is also optimized for the rare instances when the static scalar network is insufficient. When routers transmit scalar data, the high bits of data buffers are clock-gated, reducing energy as shown in (f). Figure 3.9 summarizes the area breakdown of all the network configurations that we evaluate.

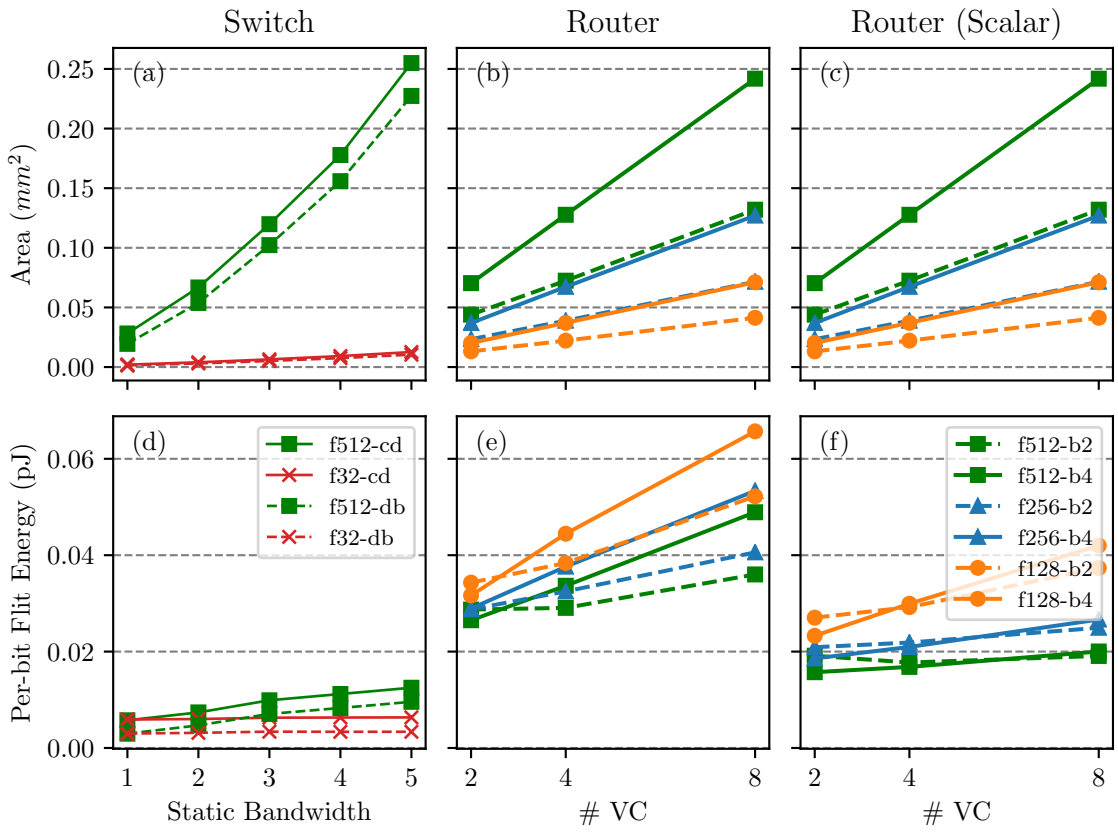


Figure 3.8: Area and per-bit energy for (a,d) switches and (b,c,e,f) routers. (c,f) Subplots (c,f) show area and energy of the vector router when used for scalar values (32-bit).

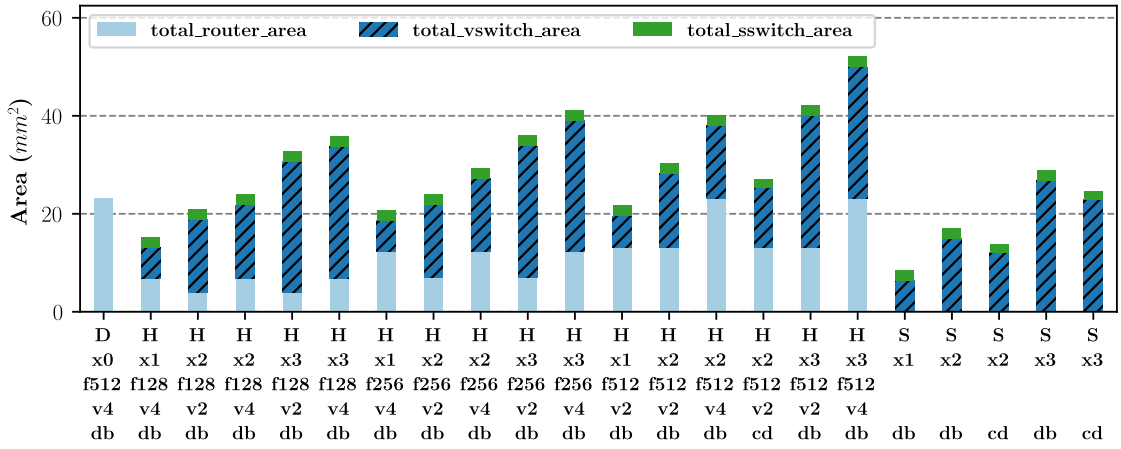


Figure 3.9: Area breakdown for all network configurations.

Notation	Description
[S,H,D]	Static, hybrid, and dynamic network
x#	Static bandwidth on vector network (#links between switches)
f#	Flit width of a router or vector width of a switch
v#	Number of VC in router
b#	Number of buffers per VC in router
[db,cd]	Buffered vs. credit-based flow control in switch

Table 3.2: Network design parameter summary.

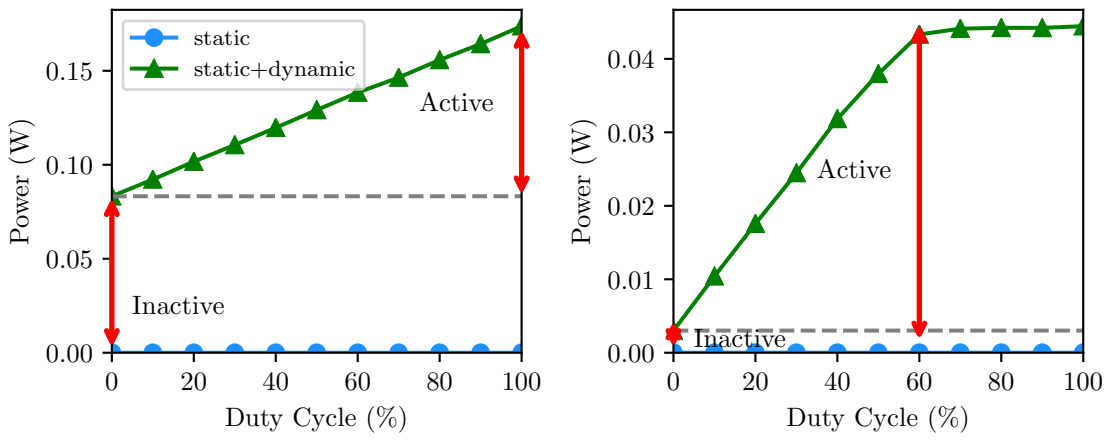


Figure 3.10: Switch and router power with varying duty cycle.

3.3.4 Network Architecture Exploration

We evaluate our network configurations in five dimensions: performance (perf), performance per network area (perf/area), performance per network power (perf/watt), network area efficiency (1/area), and network power efficiency (1/power). Among these metrics, performance is the most important: networks only consume a small fraction of the overall accelerator area and energy (roughly 10-20%). Because the two key advantages of hardware accelerators are high throughput and low latency, we filter out a network design point if it introduces more than 10% performance overhead. This is calculated by comparing to an ideal network with infinite bandwidth and zero latency.

For metrics that are calculated per application, such as performance, performance/watt, and power efficiency, we first normalize the metric with respect to the worst network configuration for that application. For each network configuration, we present a geometric mean normalized across all applications. For all of our experiments, except Section 3.3.4, we use a network size of 14×14 end-point PBs. All vector networks use a vectorization factor of 16 ($512\,\mathrm{bit}$ messages).

Bandwidth scaling with network size

Figure 3.11 shows how different networks allow several applications to scale to different numbers of PBs. For IO-bound applications (BlackScholes and TPCHQ6), performance does not scale with additional compute and on-chip memory resources. However, the performance of compute-bound applications (GEMM and SGD) improves with increased resources, but plateaus at a level that is determined by on-chip network bandwidth. This creates a trade-off in accelerator design between highly vectorized compute PBs with a small network—which would be underutilized for non-vectorized problems—and smaller compute PBs with limited performance due to network overhead. For more finely grained compute PBs, both more switches and more costly (higher-radix) switches must be employed to meet application requirements.

The scaling of time-scheduled accelerators (bottom row) is much less dramatic than

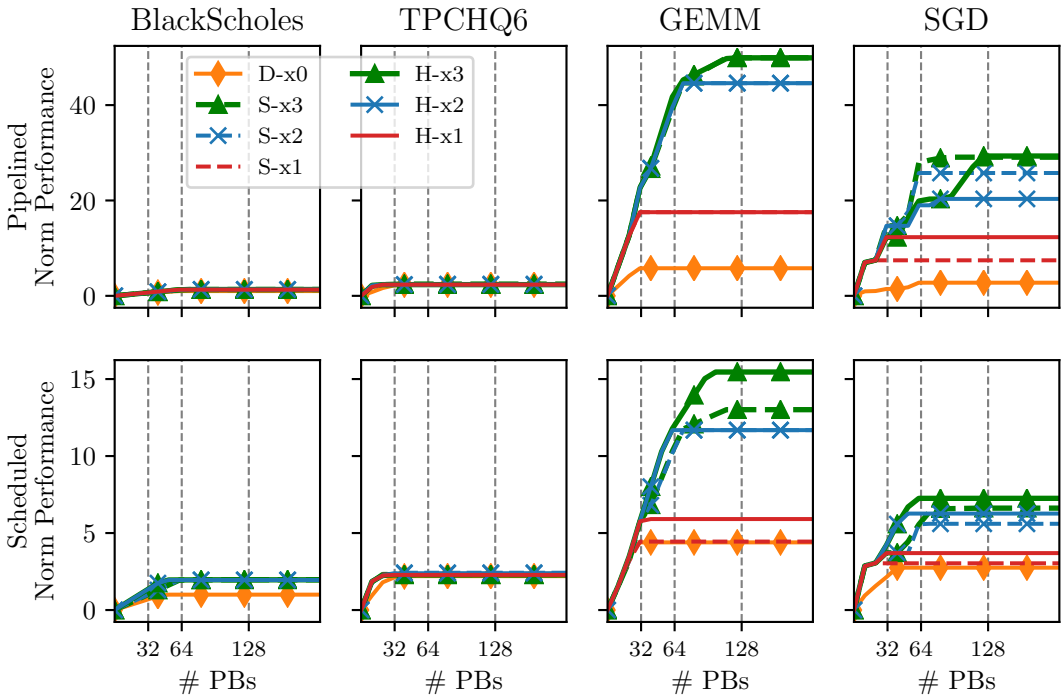


Figure 3.11: Performance scaling with increased CGRA grid size for different networks.

that of deeply pipelined architectures (top row). Although communication between PBs in these architectures is less frequent, the scheduled architecture must use additional parallelization to match the throughput of the pipelined architecture; this translates to larger network sizes.

For pipelined architectures, both hybrid and static networks provide similar scaling with the same static bandwidth: the additional bandwidth from the dynamic network in hybrid networks does not provide additional scaling. This is mostly due to a bandwidth bottleneck between a PB and its router, which prevents the PB from requesting multiple elements per cycle. Hybrid networks tend to provide better scaling for time-scheduled architectures; multiple streams can be time multiplexed at each ejection port without losing performance.

Bandwidth and flow control in switches

In this section, we study the impact of static network bandwidth and flow control mechanism (per-hop vs. end-to-end credit-based). On the left side of Figure 3.13, we show that increased static bandwidth results in a linear performance increase and a superlinear increase in area and power. As shown in Section 3.3.4, any increase in accelerator size must be coupled with increased network bandwidth to effectively scale performance. This indicates that network overhead will increase with the size of an accelerator.

The right side of Figure 3.13 shows that, although credit-based flow control reduces the amount of buffering in switches and decreases network area and energy, application performance is significantly impacted. This is the result of imbalanced data-flow pipelines in the program: when there are parallel long and short paths over the network, there must be sufficient buffer space on the short path equal to the product of throughput and the difference in latency. Because performance is our most important metric, credit-based flow control is not feasible, especially because the impact of bubbles increases with communication distance, and therefore network size.

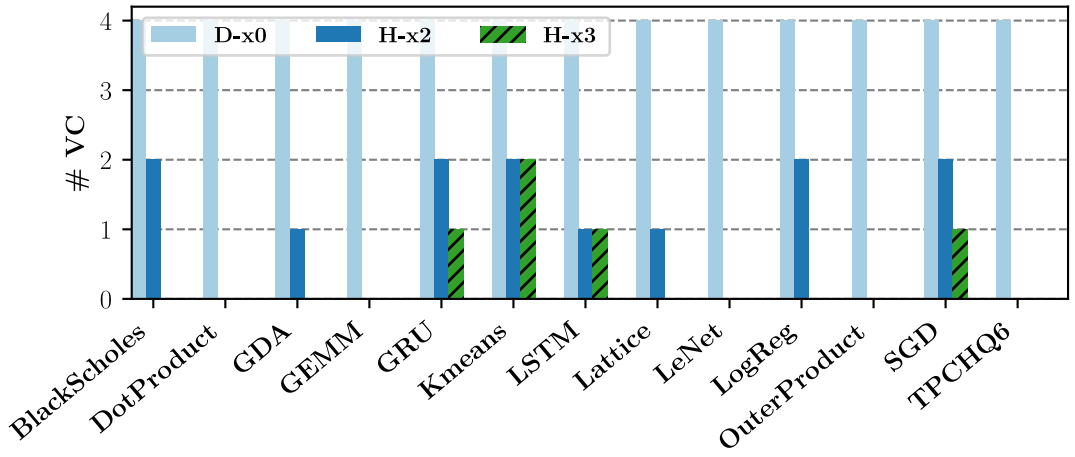


Figure 3.12: Number of VCs required for dynamic and hybrid networks. (No VCs indicates that all traffic is mapped to the static network.)

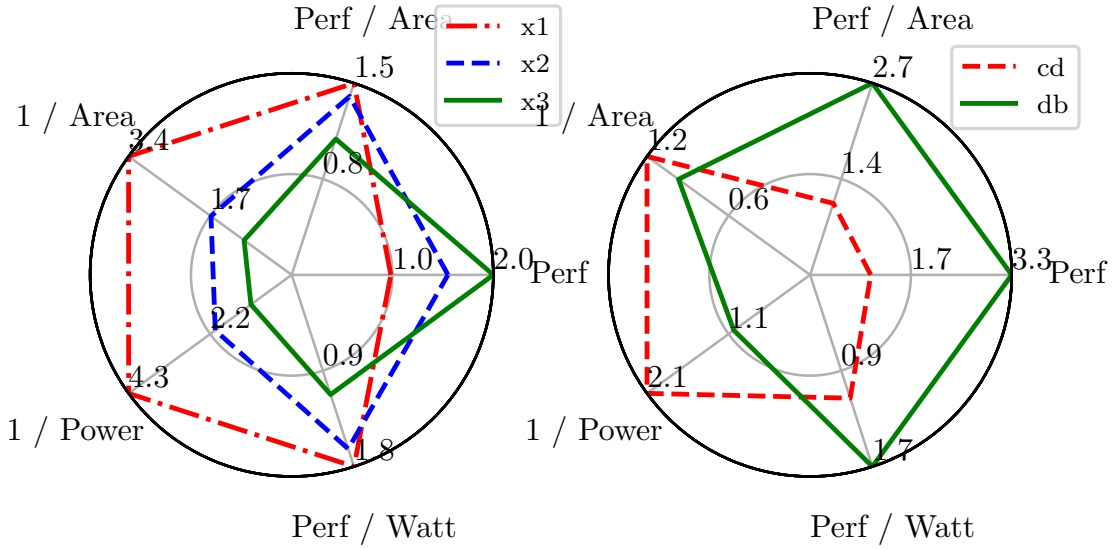
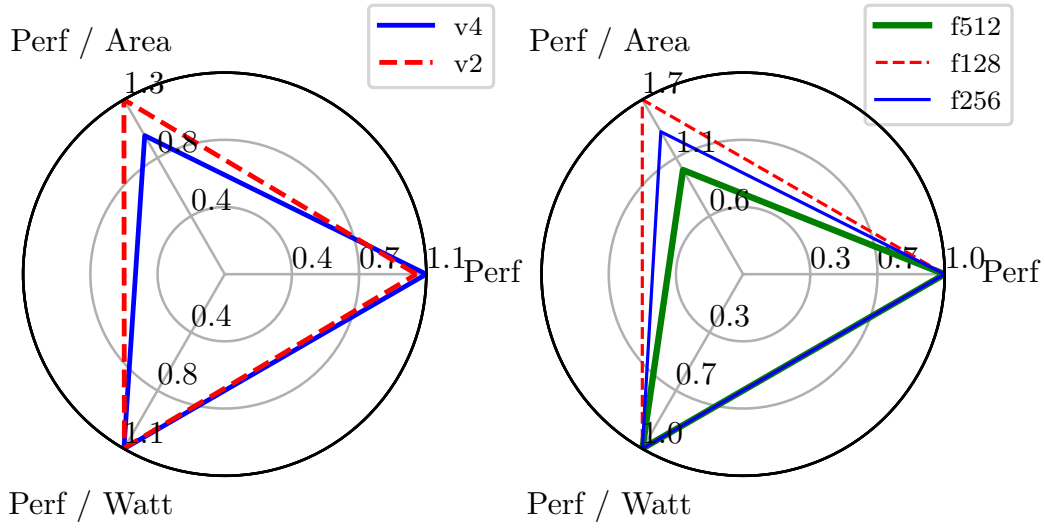


Figure 3.13: Impact of bandwidth and flow control strategies in switches.



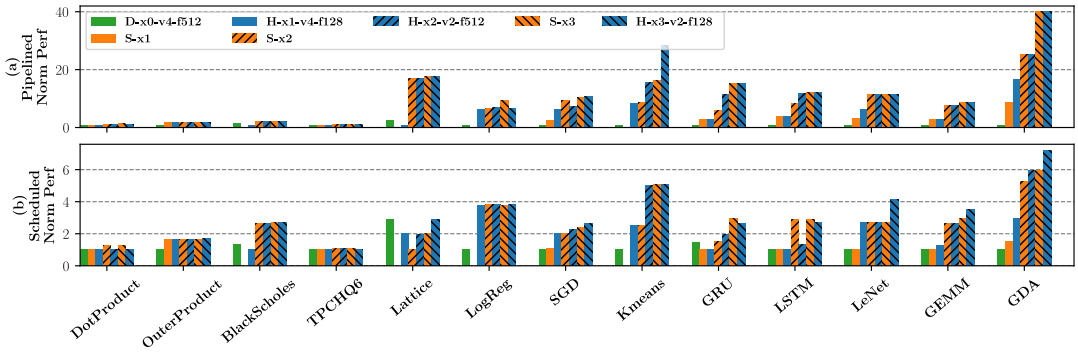


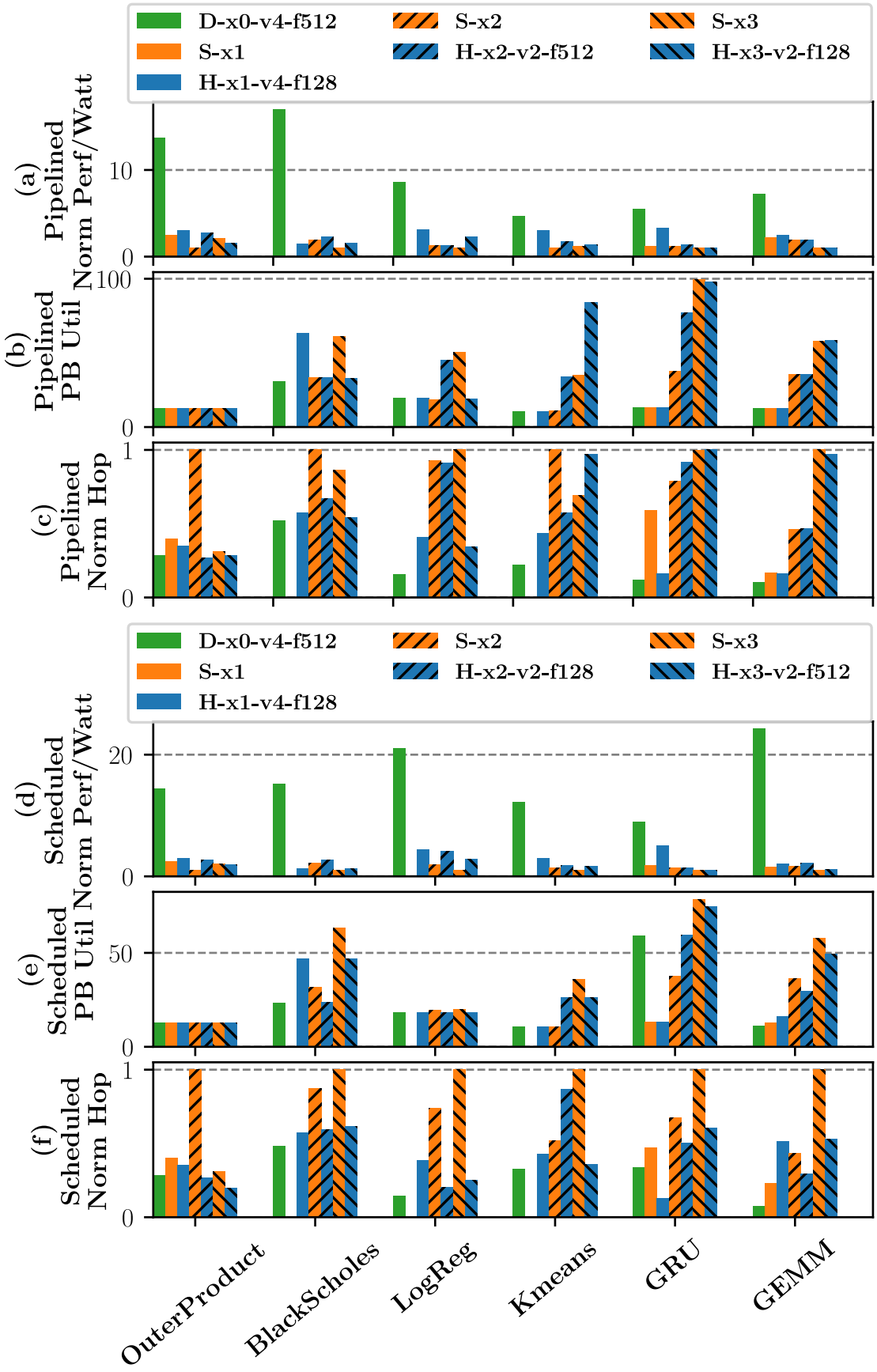
Figure 3.16: Normalized performance for different network configurations.

VC count and reduced flit width in routers

In this experiment, we study the area-energy-performance trade-off between routers with different VC counts. As shown in Section 3.3.3, using many VCs increases both network area and energy. However, using too few VCs may force roundabout routing on the dynamic network or result in VC allocation failure when the network is heavily utilized. Nonetheless, the left side of Figure 3.14 shows minimal performance improvement from using more VCs.

Therefore, for each network design, we use a VC count equal to the maximum number of VCs required to map all applications to that network. Figure 3.12 shows that the best hybrid network configurations with 2x and 3x static bandwidth require at most 2 VCs, whereas the pure dynamic network requires 4 VCs to map all applications. Because dynamic network communication is infrequent, hybrid networks with fewer VCs provide both better energy and area efficiency than networks with more VCs, even though this constrains routing on the dynamic network.

We also explore the effects of reducing dynamic network bandwidth by using smaller routers; as shown in Section 3.3.3, routers with smaller flits have a much smaller area. Ideally, we could scale static network bandwidth while using a low-bandwidth router to provide an escape path and reduce overall area and energy overhead. The right side of Figure 3.14 shows that, for a hybrid network, reducing flit width improves area efficiency with minimal performance loss.


D-x0-v4-f512
S-x1
H-x1-v4-f128
S-x2
H-x2-v2-f512
S-x3
H-x3-v2-f128
(a) Pipelined Norm Perf/Watt
(b) Pipelined PB Util
(c) Pipelined Norm Hop
D-x0-v4-f512
S-x1
H-x1-v4-f128
S-x2
H-x2-v2-f128
S-x3
H-x3-v2-f512
(d) Scheduled Norm Perf/Watt
(e) Scheduled PB Util
(f) Scheduled Norm Hop
OuterProduct
BlackScholes
LogReg
Kmeans
GRU
GEMM

Static vs. hybrid vs. dynamic networks

Figure 3.16 shows the normalized performance for each application running on several network configurations. For some applications, the bar for S-x1 is missing; this indicates that place and route failed for all unrolling factors. For DRAM-bound applications, the performance variation between different networks is trivial because only a small fraction of the network is being used. In a few cases (Kmeans and GDA), hybrid networks provide better performance due to slightly increased bandwidth. For compute-bound applications, performance primarily correlates with network bandwidth because more bandwidth permits a higher parallelization factor.

The highest bandwidth static network uses the most PBs, as shown in Figures 3.17(b,e), because it permits more parallelization. It also has more data movement, as shown in (c,f), because PBs can be distributed farther apart. Due to bandwidth limitations, low-bandwidth networks perform best with small unrolling factors—they are unable to support the bisection bandwidth of larger program graphs. This is evident in Figures 3.17(b,e), where networks D-x0-v4-f512 and S-x2 have small PB utilizations.

With the same static bandwidth, most hybrid networks have better energy efficiency than the corresponding pure static networks, even though routers take more energy than switches to transmit the same amount of data. This is a result of allowing a small amount of traffic to escape onto the dynamic network: with the dynamic network as a safety net, static place and route tends to converge to better placements with less overall communication. This can be seen in Figures 3.17(c,f), where most static networks have larger hop counts than the corresponding hybrid network; hop count is the sum of all runtime link traversals, normalized per-application to the network configuration with the most hops. Subplots (e,f) show that more PBs are utilized with static networks than hybrid networks. This is because the compiler imposes less stringent IO constraints on PBs when partitioning for the hybrid network (as explained in Section **??**), which results in fewer PBs, less data movement, and greater energy efficiency for hybrid networks.

In Figure 3.15, we summarize the best perf/watt and perf/area (among network configurations with <10% performance overhead) for pipelined and scheduled CGRA architectures. Pure dynamic networks are not shown because they perform poorly due to insufficient bandwidth. On the pipelined CGRA, the best hybrid network provides a 6.4x performance increase, 2.3x better energy efficiency, and a 6.9x perf/area increase over the worst network configuration. The best static network provides 7x better performance, 1.2x better energy efficiency, and 6.3x better perf/area. The hybrid network gives the best perf/area and perf/watt, with a small degradation in performance when compared to the static network. On the time-scheduled CGRA, both static and hybrid networks have an 8.6x performance improvement. The hybrid network gives a higher perf/watt improvement at 2.2x, whereas the static network gives a higher perf/area improvement at 2.6x. Overall, the hybrid networks deliver better energy efficiency with shorter routing distances by allowing an escape path on the dynamic network.

Chapter 4

Compiler

In this section, we introduce the compiler framework—SARA—that targets Plasticine architecture from high-level programs described in the Spatial language.

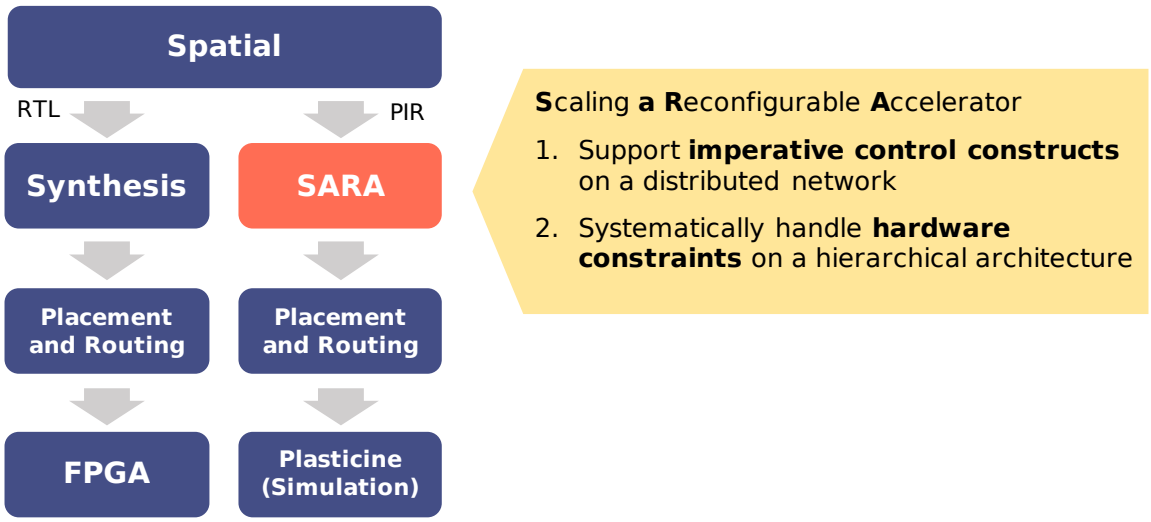


Figure 4.1: Spatial Stack

4.1 High-Level Compiler

Although SARA takes Spatial as front-end, the compilation techniques in SARA can be equally applied to other imperative langauges with nested loop constructs, such as C-based high-level syntehsis language, the backend of Halide IR, and other DSLs at the similar abstraction level. Using Spatial as our front-end language has the advantage that the language is designed for reconfigurable hardware, such that it can express valid execution schedule that can be exploit by spatial architectures natively in the language.

The data structure reprsented most exisitng imperative languages IR often marries to the CPU's virtual memory model. LLVM-based compilers, for example, treats data collections as pointers to a shared global address space, without distinguising what data are stored on-chip vs. off-chip. This is CPU provides the memory abstraction that any data within the global address space can be equally accessed and the hardware implicitly manage the data movement between on and off-chip access by brining Accelerators on the other hand, has explicitly managed on-chip scratchpads, that is not a cached version of main memory data implicitly managed by the hardware. The idea is to have the algorithm, which has better understanding of the data characteristics, to explicitly control what data

gets moved on and off-chip to maximize locality. Other important static analysis in synthesis compilers, such as banking analysis, requires the compiler to have the global view all access patterns on a data structure. Therefore, modeling the data structures as disjoint memory space is much more suitable than pointers to a shared memory for reconfigurable architectures.

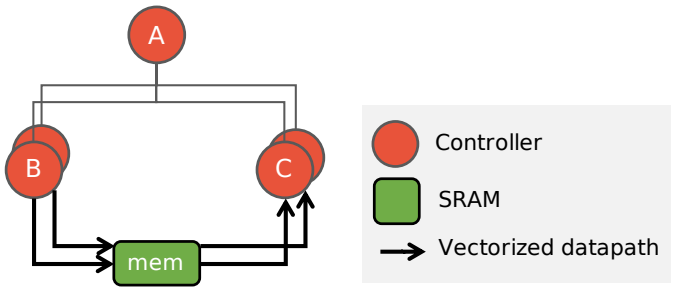
Spatial currently does not capture procedure calls in the IR; all functions are inlined and does not share resources. A future work is to use synchronization mechanism in SARA to implement true procedure call on a spatial architecture.

Without lost of generality, we use python-style pseudo code to represents the front-end programming abstraction the the rest of our discussion.

The input to SARA is the backend of the Spatial IR, which is an control hierarchy after loop unrolling. The controller at each level of the hierarchy corresponds to a control primitive, such as a loop, or a branch statement. A basic block is attached to each *inner most* controller including instructions and memory accesses to user declared data-structures. Figure 4.2 shows an example of a program and a schematic Spatial output IR. If the program has instructions outside a outer loop, Spatial automatically inserts a *unit* controller that wraps the floating instructions.

```
mem = OnchipArray(dims=[100])
A: for i in range(A,par=2):
    B: for j in range(B,vec=16):
        addr1, data1 = ...
        mem(addr1) = data1 # W1
    C: for k in range(C,vec=16):
        addr2 = ...
        data2 = mem(addr2) # R1
        ...
```

(a) Pseudo Spatial example



(b) Schematic Spatial IR

Figure 4.2: Pseudo example of SARA's front-end language. (a) shows the an example of SARA's front-end language. The actual front-end Spatial is a Scala-embeded DSL. For simplicity and generality, we use a python style pseudo code to show an language with similar abstraction. The 'par' keyword indicates outer loop unrolling factor, and the 'vec' keyword is followed by a inner-loop vectorization factor. When an iterator is vectorized, instructions using the vectorized iterator is automatically vectorized as well. When unrolling the outer loop `A`, the enclosed loop body and next-level control hierarchy are duplicated, as suggested in (b). Each loop in (a) corresponds to a controller in (b). The inner most controllers `B` and `C` each contain a basic block within instructions within the inner most loops.

4.2 Low-level Native Programming Interface of Plasticine

Similar to FPGA high-level synthesis tools, SARA provides a high-level imperative programming abstraction and synthesizes the program to execute on a reconfigurable accelerator. Targeting an RDA, however, is much more challenging than targeting FPAGs due to RDA's stringent mapping constraints. Unlike FPGAs, RDAs cannot map arbitrary RTL functionality.

Taking Plasticine as an example, the hardware has collection of memory tiles (PMUs) and compute tiles (PCUs). The mesh global network can be statically configured to connect any tiles with guaranteed in-order transmission of packets over arbitrary network latency.

Pattern Compute Unit (PCU)

As the major compute work horse of the architecture, a PCU contains a 6-stage SIMD pipeline with 16 SIMD lanes. Unlike a processor core, the PCU can only statically configure six vector instructions throughout the entire execution. Additionally, the six instructions must be branch-free in order to be fully pipelined across stages. At runtime, the SIMD pipeline executes the same set of instructions over different input data. Software can configure the SIMD pipeline to depend on a set of input streams and produce a set of output streams. There are three types of streams—single-bit control streams, 32-bit word scalar streams, and 16-word vector streams—corresponding to three types of global networks. Execution of the PCU is triggered by the arrival of its input dependencies and back pressured by the downstream buffers. The PCU also contains configurable counters, which can be chained to produce the values of nested loop iterators used in the datapath.

We refer to the program graph that can be executed by the SIMD pipeline as a **compute context** or simply **context**. A context includes the branch-free instructions mapped across SIMD stages, the input and output streams, and associated counter states and control configurations. Figure 4.3a shows example of simplified pseudo assembly code to program a PCU context. The control signals of the configurable counters, such as counter saturation or **counter done** signals, can be used to dequeue and enqueue the input and output streams,

```
# scalar and vector input streams
bufferA = VecInSteram()
bufferB = ScalInSteram()
bufferC = VecInSteram()
outputD = VecOutSteram()

i = Counter(min=0,max=3,stride=1)
j = Counter(min=0,max=3,stride=1)
chain = CounterChain(i,j)

a = bufferA.deq(when=j.valid())
b = bufferB.deq(when=i.done())
c = bufferC.deq(when=j.done())

forward(stage=0, dst="PR0", src=c)
# b is broadcasted to all lanes
stage(0,"mul",dst="PR1", a, b)
stage(1,"add",dst="PR2",oprd=["PR0","PR1"])
forward(2,dst="PR1",src="PR1")
forward(3,dst="PR1",src="PR1")
forward(4,dst="PR1",src="PR1")
forward(5,dst="PR1",src="PR1")
outputD.enq(data="PR1", when=j.valid())
```

(a) Declarative configuration

```
with Context() as ctx:
    bufferA = VecInSteram()
    bufferB = ScalInSteram()
    bufferC = VecInSteram()
    outputD = VecOutSteram()

    b = bufferB.deq()
    for i in range(0, 3, 1)
        c = bufferC.deq()
        for j in range(0, 3, 1)
            a = bufferA.deq()
            expr = a * b + c
            outputD.enq(data=expr)
```

(b) Imperative configuration

Figure 4.3: Pseudo PCU configuration. (a) shows the simplified declarative-style configuration for the SIMD pipeline context in a PCU. Each PCU context can produce a set of output streams as a function of input streams and counter values. By configuring when each stream is enqueued and dequeued using signals from the chained counters, these streams are effectively read and written within different loop bodies. (b) shows the equivalent configuration in a imperative-style on the right. On the left, the `valid` signal of the inner most counter `j` is high whenever the context is enabled. The context is implicitly triggered whenever its input streams `streamA`, `streamB`, and `streamC` are non-empty and output stream `outputD` is ready. The declarative configuration also specifies a pipeline register (PR) to holds output of each stage and explicitly forward live variables across stages. For simplicity, these details are omitted in the imperative-style on the right and **??** gives more details on register allocation for the SIMD pipeline.

respectively. The counter bounds (i.e. min, max, and stride) can also be data-dependent using values from the scalar input streams. Compared to other data-flow architectures, the data-flow engine in Plasticine is more flexible in that it allows dynamic enqueue and dequeue window for its input and output streams. *This feature enable Plasticine to support complex control hierarchy, suchas non-perfectly nested loops and branch statements, across contexts even though individual contexts can only executes instructions that are control-free*.

Figure 4.3b represents the effective execution achieved in an imperative-style program. For simplicity, we will use the imperative-style configuration in the later discussion. However, it is important to notice that there is a one-to-one translation from the declarative-style configuration to the imperative-style, but not in the reverse way. For example, the loops needs to be perfectly nested (only accesses can occur in the outer loops). **??** gives more details on operations supported by the PCU and programming restrictions in the imperative programming style.

There is no global scheduler to orchestrate the execution order among contexts—the execution is purely streaming and data-flow driven. The only way to order the execution of two contexts without a data-dependency is to introduce a control **token** between two contexts acting like a dummy data-dependency. This restriction eliminates the need of long-travelling wires and communication hot spot caused by a centralized scheduler, which again improves the clock frequency and scalability of the architecture.

Pattern Memory Unit (PMU)

PMUs hold all distributed scratchpads available on-chip. Each PMU contains 16 SRAM banks with 32-word access granularity. The PMU also contains pipeline stages specialized for address computation. Unlike SIMD pipelines in PCUs, these pipeline stages are non-vectorized and can only perform integer arithmetics. The address produced by the address pipeline is broadcasted to 16 banks with a configurable offset added to each bank. In contrast to PCU SIMD pipeline that has to be programmed atomically, the address pipeline stages within PMUs can be sliced into a write and a read context, triggered independently. In addition to compute stages, resources such as I/O ports, buffers, and counters are also

shared across contexts. It is the software's responsibility to make sure total resources consumed by all contexts does not exceed resource limits of the PMU. All contexts within PMU have access to the scratchpads with unprotected order. For instance, to restrict a read context to access the scratchpad after the write context, the software must explicitly allocate a token from the write context to the read context. SARA automatically generates required control tokens across context such that the memory access order is consistent to the one from high-level program order.

Unlike most accelerators at this scale, Plasticine does not have any shared global on-chip memory. This design dramatically improve the memory density and scalability of the architecture by eliminating hardware complexity and overhead to support complex cache coherence protocol. On the other hand, however, the burden of maintaining consistent view of a logical memory mapped across distributed scratchpads is left to the software. The software must explicitly configure synchronizations across PCUs and PMUs, taking into account that the network can introduce variable latency on both control and data path. One of the major contribution of SARA is to hide these synchronization burden from the programmer and still provide a programming abstraction of logical memories with configurable bandwidth and arbitrary capacity that fits on-chip.

DRAM Interface

The Plasticine architecture provides access to four DDR channels on the two sides of the tiled array, as shown in Figure 2.4. Each side has a column of DRAM address generator (DAG) specialized to generate off-chip requests. Like compute pipeline in PMUs, the pipeline in DAG is also non-vectorized with integer arithmetics. Each DAG can generates a load or a store request streams to the off-chip memory. All streams can access the entire address space of the DRAM, with in-order response within each stream and no ordering guaranteed across streams. To provide off-chip memory consistency, SARA allocates synchronizations across request generator and response receiver contexts to preserve memory ordering specified in the program.

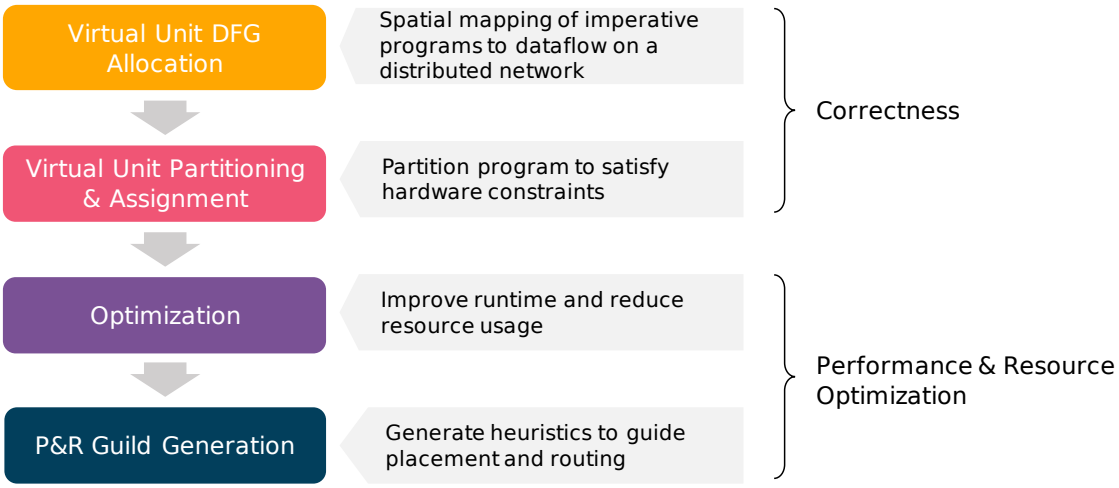


Figure 4.4: SARA Compiler Flow

4.3 Compiler Overview

In this section, we describe a systematic approach to compile applications described in an imperative front-end language with into a purely declarative and distributed dataflow graph that can run on Plasticine. Figure 4.4 shows the compilation flow to perform this translation between the two paradigms. At high-level, SARA allocates distributed on-chip resources to execute the program in spatially parallelized and pipelined fashion. SARA automatically generates synchronization across distributed units to provide strong memory consistency for a imperative program, on an architecture that does not guarantee memory ordering across multiple access streams. This synchronization is purely point-to-point introducing minimum performance overhead, which maximizes scalability of the architecture. SARA further virtualizes resources allocation and hides the underlying resource constrains on this hierarchical architecture from the programmers.

First, SARA takes the input Spatial IR and performs the **virtual allocation**. A virtual unit (VU) is our intermediate representation that captures computations that will be mapped onto a physical unit (PB), such as a PCU and PMU. Each VU can contain multiple contexts. However, a VU can be mapped to a PU only if the aggregated resource usage of all

contexts in the VU can fit in the PU. The PU can also limits the maximum number of contexts it can support, and has resources that cannot be split across contexts. Additionally, messages across VUs are mapped across the global network, and must tolerate arbitrary amount of network latencies; messages within a single VB across contexts takes only a single cycle. In the first phase, SARA generates p2p synchronizations across contexts such that the execution is the same as if there are centrolized scheduler scheduling each level of the nested control hierarchy in the original program. The allocation phase generates a virtual unit data-flow graph (VUDFG) with appropriate synchronizations, such that the on-chip distributed execution produces the same result as if the program is sequentially executed.

At the end of the allocation phase, a virtual unit can consume as much resource as the program requires. The second phase is **physical allocation**, where SARA assign each VU to a PU that processes the required resources. If no PU can execute a VU, SARA tries to partition the VU into multiple VUs to eliminate constraint violation. If there is insufficient PU or the VU cannot be partitioned, the mapping process fails with appropriate hint to the programmer for the limiting resources.

Throughout the first two phases, SARA introduces various **optimizations** that either reduce the resource cost of the VUDFG, or alleviates potential performance bottleneck in streaming pipelined execution. After all VU fits in at least one type of PU, SARA performs a global optimization that merges small VUs into a larger VU to reduce resource fragmentations.

At the end the previous phase, each VB is tagged with a PB type that can fit the VB, and it is up to the **placement and routing** (**PaR**) phase to determine where the VB will be finally placed. SARA can perform static analysis on the traffic pattern and generate heuristics for PaR to reduce runtime congestion.

In the following sections, Section 4.4 describes conversion from imperative paradigm with nested control hierarchy to the distributed streaming data-flow execution. Section 4.5 details program-partitioning passes that decompose program over distributed resources. Section 4.6 enumerates several optimizations in SARA, and **??** discuss about PaR and heuristic generation.

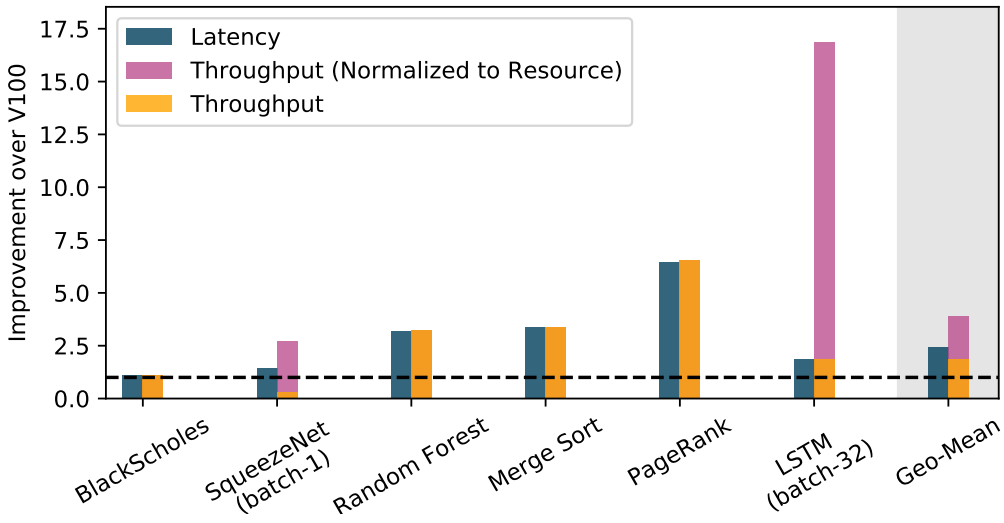


Figure 4.5: Plasticine's latency and throughout improvement over V100 GPU. The evaluated Plasticine architecture has area footprint of 352mm^2 at 28nm. V100 GPU has area footprint of 815mm^2 at 12nm. Both platforms have the same off-chip bandwidth at 1TB/s with HBM technology. Yellow and blue bars show the raw measured speedup in throughput and latency, respectively. To account for the resource discrepancy, the pink bar shows the normalized throughput for compute-bound application–SqueezeNet and LSTM, which scales performance with additionally on-chip resources.

```
1 mem = zeros(N)
2 a = rand(N)
3 b = rand(N)
4 for i in range(1, N):
5     tmp = a[i] * b[i]
6     mem[i] = tmp + i
```

(a) Input program

```
1 mem = zeros(N)
2 a = rand(N)
3 b = rand(N)
4 tmp = zeros(N)
5 for i in range(1, N):
6     tmp[i] = a[i] * b[i]
7 for i in range(1, N):
8     mem[i] = tmp[i] + i
```

(b) Loop Fission

```
1  mem = zeros(N)
2  a = rand(N)
3  b = rand(N)
4  tmp = queue()
5  @concurrent
6  for i in range(1, N):
7      tmp.enq(a[i] * b[i])
8  @concurrent
9  for i in range(1, N):
10     mem[i] = tmp.deq() + i
```

(c) Loop Division

Figure 4.6: (b) and (c) shows the output of loop fission and loop division of the input program (a), respectively. In (b), the first loop is executed entirely before executing the second loop. The intermediate result `tmp` is materialized into an array with the same size as the loop range. In (b), the two loops can execute concurrently. The intermediate result is materialized into a queue. For each iteration, a loop can execute only if all of its queues are non-empty. The second loop can execute as soon as `tmp` receives the first element.

4.4 Imperative to Streaming Transformation

4.4.1 Loop Division

Reviewing the front-end and back-end abstraction of SARA, an obvious gap between the two abstractions is that the front-end imperative language can contain arbitrary nested control hierarchy, whereas the hardware compute engine can only execute perfectly nested loops. To address this issue, we introduce a new type of loop transformation—loop division—for streaming reconfigurable accelerators. Similar to loop fission, loop division breaks a single loop into multiple loops. The difference is that loop fission generates a sequence of sequentially executed loops, whereas loop division generates loops executing *concurrently*. Additionally, loop fission materializes the intermediate results across fissioned loops into arrays, while loop division use queue to communicate across loops. Each loop generated from loop division can only execute if all of their input queues is not empty. Figure 4.6 gives an example of loop fusion vs. loop division.

When executing loop division on a single-threaded CPU, the CPU must context switching between the concurrent loops and executing the one with cleared input dependencies. Like loop fission, loop division is likely worsen the performance on a processor architecture,

```
mem = rand(N)
for i in range(1, N):
    tmp = mem[i-1] + i
    mem[i] = tmp
```

(a) Input program

```
mem = rand(N)
tmp = zeros(N)
for i in range(1, N):
    tmp[i] = mem[i-1] + i
for i in range(1, N):
    mem[i] = tmp[i]
```

(b) Invalid Loop Fission

```
mem = rand(N)
tmp = queue()
# Initialize queue with
# 1 dummy element
backward = queue(init=1)
@concurrent
for i in range(1, N):
    tmp.enq(mem[i-1] + i)
    backward.deq()
@concurrent
for i in range(1, N):
    mem[i] = tmp.deq()
    backward.enq(0)
```

(c) Loop Division

Figure 4.7: Example of Loop Fission vs. Loop Division

as the memory footprint of the intermediate result `tmp` increases from $O(1)$ to $O(N)$. On RDAs, the devisied loops are executing concurrently in streaming pipelined fashion. The size of the `tmp` can be limit to a small fixed size, efficiently implemented with a hardware FIFO. Although loop transformations are generally optimizations on CPUs, loop division is a required transformation to converts an infeasible program to a feasible one for Plasticine.

Loop fission is not always safe, as it may alters the execution order of the program. order. Loop division, on the other hand, does not change the underlying data-dependency and is always safe. To achieve this, loop division needs to introduce additional dummy data dependency to enforce the correct execution. Figure 4.7 gives an example of an invalid loop fission and a correct loop division. **??** gives more detail on how the dummy data-dependency are generated automatically.

4.4.2 Virtual Context Allocation

As a start, SARA allocates one virtual memory to hold each on-chip data structure, and one context to execute each basic block within the inner most controllers. A basic block maps naturally to a context, as instructions within a basic block are control-free. Next, SARA makes a copy of all controllers enclosing the basic block in the corresponding context; these controllers are later converted to counters and control configurations supported by the hardware. Figure 4.8 shows an example of the context allocation. With these controllers,

```
mem = array(dims=[100])
A: for i in range(A):
    B: for j in range(B):
        addr1, data1 = ...
        mem(addr1) = data1 # W1
    C: for k in range(C):
        addr2 = ...
        data2 = mem(addr2) # R1
        ...
```

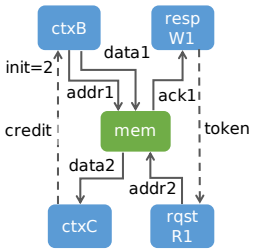
(a) Pseudo input example

```
mem = VirtualMemory(dims=[100])

with Context() as ctxB:
    A: for i in range(A):
        B: for j in range(B):
            addr1, data1 = ...
            mem(addr1) = data1 # W1

with Context() as ctxC:
    A: for i in range(A):
        C: for k in range(C):
            addr2 = ...
            data2 = mem(addr2) # R1
            ...
```

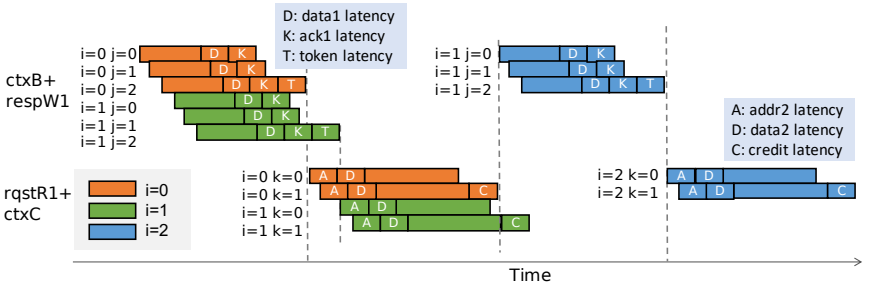
(b) Context allocation

```
mem = VirtualMemory(dims=[100])
token = ControlStream()
credit = ControlStream(init=2)

with Context() as ctxB:
    A: for i in range(A):
        B: for j in range(B):
            addr1, data1 = ...
            mem.addr.enq(addr1)
            mem.data.enq(data1)
        credit.deq()
with Context() as respW1:
    A: for i in range(A):
        B: for j in range(B):
            ack1 = mem.ack.deq()
        token.enq()

with Context() as rqstR1:
    A: for i in range(A):
        C: for k in range(C):
            addr2 = ...
            mem.reqst.enq(addr2)
        token.deq()
with Context() as ctxC:
    A: for i in range(A):
        C: for k in range(C):
            data2 = mem.resp.deq()
            ...
        credit.enq()
```

(c) Request and response division



(d) Context Graph



(e) Timing on Plasticine

Figure 4.8: Lowering example. SARA allocates one context per basic block for `B` and `C`, shown in (b). Outer controller `A` is duplicated in both `ctxB` and `ctxC`. (c) SARA seperates out requestion generator `rqstR1` from `ctxB` for `R1` and response receiver from `ctxC` `respW1` for `W1`. The resulting dataflow graph is shown in (d). To enforce the forward data-dependency between `W1` and `R1`, SARA allocates a forward token between `W1`'s response receiver `respW1` and `R1`'s request generator `rqstR1`; to enforce the loop-carried WAR dependency between `R1` and `W1`, SARA allocates a backward token (`credit`) between `R1`'s response receiver `ctxC` and `W1`'s request generator `ctxB`. The backward credit is initialized with two elements because `mem` is double-buffered. On the writer side, a forward `token` is a produced and a backward `credit` is consumed every `B` iterations; on the reader side, a forward `token` is consumed and a backward credit is produced every `C` iterations. The resulting timing of the execution is shown in (e).

contexts can repeat execution for expected number iterations. However, data-structures written and read by different contexts are accessed in random order. The insight is that *as long as all contexts accessing a shared memory with expected program order, the final result is identical to a sequentially executed program*. Unlike traditional out-of-order execution, where hardware and compiler look for independent instructions to execute concurrently, SARA starts with executing *all* basic blocks in a program in *concurrent* contexts. Next, SARA introduces synchronizations to maintain consistent access order as expected by the program *only* among contexts accessing a shared memory. This way, SARA introduces minimum p2p synchronizations among small groups of contexts; contexts accessing different memories are naturally parallelized without impacting the final output.

4.4.3 Control Allocation

Starting with all contexts execute in parallel, SARA introduces **control token**s across contexts to serialize their execution order based on the program order. This control token is no different from a regular data-dependency and can be viewed as an access grant to the shared memory across contexts. By controlling *where*, *how*, and *when* to pass the token, SARA is able to maintain a consistent update ordering between the pipelined and parallelized contexts that access the shared memory.

We refer to an memory access appeared in the input IR as a *declared access*, as supposed to memory accesses executed at runtime. `W1` and `R1` are examples of two declared accesses in Figure 4.8a In the rest of this section, we will walk through the detail on how SARA allocates control tokens to maintain sequential consistency on Plasticine.

Where. During control allocation, SARA examines all declared accesses of a memory and checks for dependency across these accesses. SARA only allocate resource to synchronize two contexts if they contain declared accesses that can potentially interfere. Whether two declared accesses interfere depends on

- the type of accesses (read vs. write)

Data structure	Memory type
array (fit on-chip)	SRAM
array (not fit on-chip)	DRAM
scalar variable	register
queue	FIFO

Table 4.1: Mapping between user declared data-structure to underlying hardware memories. Programmers explicitly specify the desired hardware type inside Spatial. In other languages, this table specifies a mapping between software data-structures and hardware memory types on Plasticine.

Memory type	DRAM	SRAM	FIFO	Register
read-after-read (RAR)	✗	✓	✓	✓
read-after-write (RAW)	✓	✓	✓	✓
write-after-read (WAR)	✓	✓	✓	✓
write-after-write (WAW)	✓	✓	✓	✓

Table 4.2: Interference table for whether two accesses interfere for each memory type.

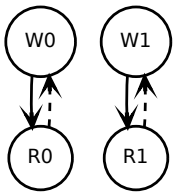
- the type of the memory (e.g. SRAM, DRAM)
- and location of the declared accesses in the control hierarchy.

Table 4.1 lists hardware memories available on reconfigurable architectures and software data-structures providing similar program semantics. The type of the memory matters because they have different programming interface in the hardware. For instance, two read accesses does not interfere for DRAM because the DRAM interface permits multiple concurrent access streams through multiple DAGs[1]. mension port virtualization Therefore, from the programmer perspective, users do not need to serialize the two contexts reading the same DRAM address. SRAMs, on the other hand, have single read and write ports. Programmers must guarantee that a PMU receives read requests from a single context at any point in time for correctness. Table 4.2 shows the interference relation between different types of memory across accesses.

[1] All DAGs can access the full DRAM address space

```
1  mem = OnchipArray(dims=[100])
2  A: for i in range(A,par=2):
3      ## A0: lane 0 of loop A
4      B: for j in range(B):
5          mem(addr1) = data1 # W0
6      C: for k in range(C):
7          data2 = mem(addr2) # R0
8      ## A1: lane 1 of loop A
9      B: for j in range(B):
10         mem(addr1) = data1 # W1
11     C: for k in range(C):
12         data2 = mem(addr2) # R1
```

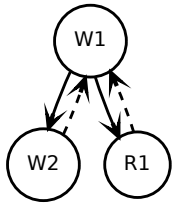
(a) Pseudo example



(b) Dependency Graph

```
1  mem = OnchipArray(dims=[100])
2  A: for i in range(A):
3      B: mem(addr1) = ... # W1
4      C: if cond:
5          D: mem(addr2) = ... # W2
6         else:
7          E: ... mem(add3) # R1
```

(c) Pseudo example



(d) Dependency Graph

Figure 4.9: Dep Graph

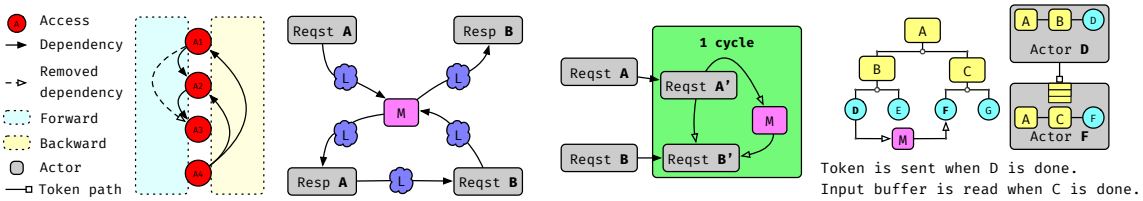


Figure 4.10: (a) Access dependency graph. (b) Synchronization of two accesses on the same memory. (c) Single-cycle special case. (d) Actors uses local states of controller hierarchy to determine when to send a token.

For every declared access, SARA checks on other accesses appeared earlier in the program order for a possible forward dependency, and later in the program order for a possible loop-carried dependency (LCD). In the example in Figure 4.8a, there is a forward data dependency between `W1` and `R1`, and a LCD between `R1` and `W1`. SARA builds a dependency graph between declared accesses for each memory. Figure 4.9 gives more examples on how SARA determines dependencies based on where these accesses are declared in the control hierarchy.

Enforcing all dependencies in the dependency graph may not be necessary, as enforcing a subset of dependencies can be sufficient to enforce the order of the whole graph. Therefore, SARA performs a transitive reduction (TR) on the graph to keep the minimum number of dependency edges that preserve the same ordering [**?**]. Since TR on a cyclic graph is NP-hard,]name perform TRs on the forward and backward LCD graphs, separately. Notice, dependencies between accesses touching different buffers of a multi-buffered memory is less rigid than accesses touching the same buffer. Therefore, we can only remove an edge if all dependencies on the equivalent path have a stronger or equivalent dependency strength than the strength of the removed edge.

How. To eliminate the round-trip overhead between the memory and the computation, SARA duplicates the local states and expressions required to generate the requests in a separate context as the one that handles the responses. For write accesses, the memory provides an acknowledgment for each request received, used by SARA for synchronization.

The request context generates requests asynchronously as soon as its data-dependencies are cleared, pushing all requests to memory until back-pressured. To order a declared access A before a declared access B SARA creates a dummy dependency between the context that accumulates the response of access A ($resp_A$) and the context that generates requests for access B ($reqst_B$) (Figure 4.10(b)). To enforce LCD from access B to access A, SARA introduces a token from $resp_B$ to $reqst_A$, and initializes the token buffer (input buffer receiving the token) with one element to enable the execution of the first iteration. If the LCD is on a multi-buffered memory, the LCD token is initialized with the buffer depth number of elements to enable A for multiple iterations before blocked by access B.

These are general schemes we use on any types of memory (including DRAM and on-chip memories) with unpredictable access latency.

Special Case: Single-Cycle Access For a memory with guaranteed *single-cycle* access latency, such as registers and statically banked SRAMs that are guaranteed conflict-free, we can simplify the necessary synchronization (Figure 4.10(c)). Instead of synchronizing between $resp_A$ and $reqst_B$, we allocate two stateless contexts $reqst'_A$ and $reqst'_B$ within the same VB as the accessed memory that forwards requests from $reqst_A$ and $reqst_B$, respectively. Next, we forward the token going from $reqst_A$ to $reqst_B$ to go through $reqst'_A$ to $reqst'_B$ instead, and configure the token buffer in $reqst'_B$ with the depth of one for serialized schedule and depth of M for multi-buffered schedule. We no longer need to insert the LCD token, as the stiff back pressure from the token buffer in $reqst'_B$ will enforce the expected behavior. This optimization only works if the sender and receiver of the token buffer are physically in a single VB where the memory is located. In this way, when $reqst'_B$ observes $reqst'_A$s token, $reqst'_B$ is guaranteed to observe the memory update from $reqst'_A$ because the memory also has single-cycle access latency.

When. SARA configures the contexts to generate the token using their local states at runtime. For FIFOs, the token is generated and consumed every cycle when the producer and receiver contexts are active. For register, SRAM, and DRAM the program order expects

```
1 A: for i in range(N):
2     B: for j in range(2*i, N):
3         ... = i + j
```

(a) Input program

```
1 bound = ScalarStream()
2 with Context() as ctx1:
3     A: for i in range(N):
4         bound.enq(2*i)
5 with Context() as ctx2:
6     A: for i in range(N):
7         start = bound.deq()
8         B: for j in range(start, N):
9             ... = i + j
```

(b) Context graph

Figure 4.11: (a) shows an example program with dynamic loop range. Expressions to generate the loop bounds belongs to a basic block that gets mapped to `ctx1`. SARA maps the loop bound as a data-dependnecy to context that maps the inner loop context, and configures dequeue signal of `bound` stream to counter `B.done`.

that the producer and consumer writes and inspects the memory once per iteration of their LCA controller, respectively. Since the producer and receiver both have their outer controllers duplicated in their local state, they have independent views for one iteration of the LCA controller, which is when the controller in their ancestors (that is the immediate child of the LCA controller) is completed (Figure 4.10(d)). The *done* signals of these controllers are used to produce and consume the token in contexts, independently.

4.4.4 Data-Dependent Control Flow

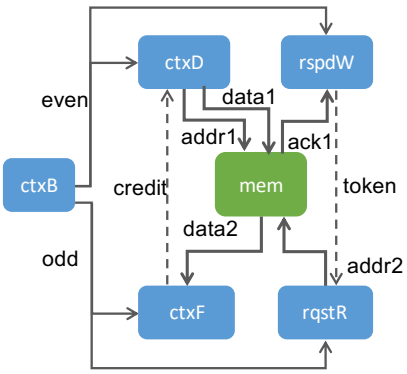
Using the synchronization discussed in Section 4.4.3, SARA can support control constructs that are typically not supported on dataflow accelerators, such as branches and while loops. To enable this, the control path of the architecture must permit data-dependency. Figure 4.12 shows an example of data-dependent loop ranges. SARA uses a context to compute the loop bound, and the loop bounds are treated as input dependency to the context mapping the loop body.

The branch condition is also treated as a data-dependent enable signal of controllers under branch clauses.

If the controller is disabled, it is considered *done* immediately. Output tokens depending on the *done* signal will be immediately sent out. For a memory written inside a branch

```
A: for i in range(A):
    B: even = i % 2 == 0
    C: if even:
        D: for j in range(D):
            addr1, data1 = ...
            mem(addr1) = data1 # W
       else:
        F: for k in range(F):
            addr2 = ...
            data2 = mem(addr2) # R
            ...
```

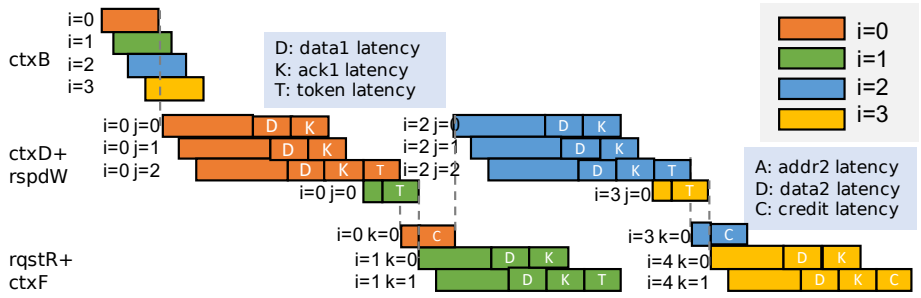
(a) Input program



(b) Dataflow graph

```
even = ScalarStream()
odd = ScalarStream()
token = ControlStream()
credit = ControlStream(init=2)
with Context() as ctxB:
 A: for i in range(A):
     B: tmp = i % 2 == 0
        even.enq(tmp)
        odd.enq(not tmp)
with Context() as ctxD:
 A: for i in range(A):
     C: if even.deq():
         D: for j in range(D):
             addr1, data1 = ...
             mem.waddr.enq(data1)
             mem.wdata.enq(data1)
        credit.deq()
with Context() as rpstW:
 A: for i in range(A):
     C: if even.deq():
         D: for j in range(D):
             pass
        token.deq()
with Context() as rqstR:
 A: for i in range(A):
     C: if odd.deq():
         F: for k in range(F):
             addr2 = ...
             mem.raddr.deq(addr2)
        token.deq()
with Context() as ctxF:
 A: for i in range(A):
     C: if odd.deq():
         F: for k in range(F):
             data2 = mem.rdata.deq()
             ...
        credit.enq()
```

(c) Context graph



(d) Context configuration

Figure 4.12: (a) shows an example program with dynamic loop range. Expressions to generate the loop bounds belongs to a basic block that gets mapped to `ctx1`. SARA maps the loop bound as a data-dependnecy to context that maps the inner loop context, and configures dequeue signal of `bound` stream to counter `B.done`.

statement and read outside the branch (with a branch miss), the writer contexts immediately sends out the token to the receiver as soon as the branch condition is resolved. With a branch hit, the controller waits until its inner controller completes before raising the *done* signal. A similar scheme is used to implement the while loop, where the while condition is a data-dependency of stop signal of controller X. The producer of the while condition also consumes its own output as an LCD. The condition is then broadcast to all other contexts under the same while loop. The *done* signal of the while loop is raised when the condition's data-dependency evaluates to true. At this point, contexts accessing memory within the while loop will send the token to access contexts outside of the while loop, and enable them to access the intermediate memory.

After all contexts and shared resources are allocated and synchronized, we simply put each context and shared resource into their own VBs. The contexts with single-cycle special case (Section 4.4.3) must be put in the same VB as the shared memory.

4.5 Resource Allocation

The output of the transformation (Section 4.4) is a VUDFG that can execute on a Plasticine with infinite-sized physical units (PUs). The *Resource Allocation* phase enforces and addresses constraint violations given the specification of the Plasticine units. At the end of this phase, SARA assigns each VU in the VUDFG graph to a PU type with required resources; the placer then takes the type assignments and determines the final placement.

Accelerators often have heterogeneity in compute resources to improve efficiency for commonly used special operations. In Plasticine, PMUs and DAGs have specialized compute pipelines for address calculation that are less capable than the compute pipeline in PCUs. However, heterogeneity tends to reduce average utilization because different applications, and even the same application with different data sizes, can vary highly in the desired ratio among different resource. A compute-bound application, for example, can heavily underutilize the DAGs and PMUs. To address this problem, SARA models the virtual to physical assignment as a constraint satisfaction problem; each VU consumes a set of resources and can only be assigned to a PU if the PU processes the required resources. Table 4.3 shows the types of resources in Plasticine's heterogeneous units. For example, special connection to off-chip memory interface is also treated as a type of resource in the DAG, which forces virtual contexts accessing DRAM to map to DAGs. On the other side, regular contexts with non-vectorized fixed-point operations can also be mapped to spare DAGs, which improves utilization.

As shown in Algorithm 1, the *resource allocation* phase contains three steps: *constraint resolution*, *global merging*, and *virtual to physical assignment*. SARA uses a VU-PU bipartite graph (`G`) to keep track of potential valid assignments between the two. Initially, `G` is initialized to a complete bipartite graph, i.e. all VUs can be assigned to all PUs.

Constraint Resolution A list of constraint pruners, each considering a set of on-chip resources, incrementally remove the VU-PU edges that violate the resource constraints. If a

```
Function alloc(V, P, pruners): /* Allocation Algorithm                    */
    Data: V: a set of VUs from the VUDFG
    Data: P: a set of all PUs on the hardware
    Data: pruners: a list of constraint pruners to check and fixes constraint
          violations
    /* Initialize a complete bipartite graph                              */
    G = new BipartiteGraph();
    G[V] = P;
    /* Constraint resolution                                              */
    prune(G, pruners);
    /* Global merging                                                     */
    merge(G);
    /* Heuristic check on whether assignment is feasible                  */
    check(G);
    /* Virtual to physical assignment                                     */
    backtracking_assign(G);

Function prune(G, pruners): /* A recursive pruning function               */
    Data: G: bipartite graph between VUs and PUs
    Data: pruners: a list of constraint pruners to check and fixes constraint
          violations
    Result: The function update G by removing VU-PU edges that violates
            constraints guarded by pruners. The function may fail and raise an
            exception.
    for pruner in pruners:
        for v in G.keys():
            for p in G[v]:
                if !pruner.fit(v,p):
                    G[v] -= p;
            if G[v].empty():
                /* Partition VU v based on resource constraints
                   registered in pruner.  Not all resources can be
                   partitioned and this step may fail.  If
                   succeeded, the function returns a new set of
                   VUs.                                                   */
                V' = pruner.partition(v);
                G' = new BipartiteGraph();
                G'[V'] = G.values();
                prune(G',pruners);
                G -= v;
                G[V'] = G'[V'];
```

Algorithm 1: Resource allocation. The bipartite graph `G` contains a bi-directional many-to-many map. `G[key]` returns a set of values, and `G[value]` returns a set of keys. `G[KeySet] = ValueSet` creates all-to-all assignment between `KeySet` and `ValueSet`.

Feature	PCU	PMU	DAG	Host Unit
Vector lane width	16	16	1	1
Fixed-point op	✓	✓	✓	✗
Float-point op	✓	✗	✗	✗
Number of stages	6	10	5	0
Number of pipeline registers	8	8	4	0
Reduction tree	✓	✗	✗	✗
# Vector FIFO	6	6	4	0
# Scalar FIFO	6	6	4	16
# Control FIFO	16	16	4	16
Scratchpad banks	0	16	0	0
Scratchpad capacity	0	256kB	0	0
MergeBuffer	✓	✗	✗	✗
Splitter	✓	✗	✗	✗
Scanner	✓	✗	✗	✗
Access to DRAM Interface	✗	✗	✓	✗
Access to Host IO	✗	✗	✗	✓

Table 4.3: MergeBuffer, Splitter, and Scanner are new hardware introduced in [**?**] and [**?**] to support database and sparsity in Plasticine.

```
Function check(G): /* Assignment feasibility check                    */
    Data: G: bipartite graph
    Result: whether assignment is possible
    /* For every value set in G                                       */
    for V in G.values().toSet():
        K = ∅;
        for v in V:
            for k in G[v]:
                if G[k] ⊂ V:
                    K += k;
        if |K| > |V|:
            return failure();
    return success();
```

Algorithm 2: Heuristic check on whether it is possible to assign all key with an value in a bipartite graph. Given there are only a few types of hardware tiles, $G.values().toSet()$ is relatively small. This algorithm roughly runs in $O(|G.keys()| \times |G.values()|)$, which is still much faster than the backtracking assignment, which has exponential runtime.

VU `v` has no mappable PU after pruning, the pruner attempts to fix the violation by decomposing the VU into multiple VUs. Not all resources can be composed and the partitioning transformation may fail. If succeeded, the partitioner generates a new set of VUs `V'`. SARA starts a new complete bipartite graph between `V'` and all resources `P`, and recursively prune on `V'`. If succeeded, the original graph `G` is updated with `V'` and their pruned resources.

Global Merging After all VUs have at least one PU in the bipartite graph, SARA triggers a global optimization that merges small VUs into a larger VU to reduce fragmentation in allocation. Each type of resource has an aggregation rule to compute how the resource usages change if two VUs are merged together. Most aggregation rules are simple, such as addition, logical or, max, or union. The in- and out-degree increases are tricker and will be detailed in Section 4.5.1.

Virtual to Physical Assignment Next, SARA performs a quick heuristic check on the bipartite graph to see if there exists a possible assignment for all VUs with sufficient PUs (Algorithm 2), and provide feedback on the limiting resources, otherwise. Finally, SARA assigns each VU to a PU type with a backtracking search on the pruned bipartite graph.

This approach can be easily extended to handle new heterogeneous tiles in the architecture by registering new types of resources with aggregation and partitioning rules. The rest of this section will focus on two types of partitioning transformations–compute partitioning in Section 4.5.1 and memory partitioning in Section 4.5.2.

4.5.1 Compute Partitioning

The *compute-partitioning* phase addresses VUs using more compute resource than any PU can provide. If a VU contains multiple contexts, SARA first moves the contexts into separate VUs. If a single context exceeds the resource limit, SARA breaks down the dataflow graph in the context into multiple contexts and put them in separate VUs. During partitioning, SARA maps each subgraph of the large dataflow graph into a new context, mirrors

Problem	Partition the dataflow graph into subgraphs such that all subgraphs satisfy the constraints of a hardware unit.
Objective	Minimize the number of partitions and connectivity across partitions.
Constraints	Each partition must not exceeds the limit on the number of • live in/out variables (I/O ports) • operations (pipeline stages), • and live variables across operations (pipeline registers), etc. No *new* cycles can be formed across partitions other than the cycles in the original dataflow graph.

Table 4.4: Formulation of the compute partitioning problem

the control states of the original context, and streams live variables in between. We can formulate the problem of how to partition in the dataflow graph as an optimization problem, shown in Table 4.4. The partitioner "fixes" the VU `v` based on a single PU specification, albeit there are many potential PUs the decomposed VU can be mapped to. Currently, we use a heuristic to select a PU type from `G[v]` right before the compute pruning as a guiding constraint for partitioning.

Because the global network is specialized to handle efficient broadcasts, the in/out-degree of a partition counts the number of unique live-in/out variables, as supposed to number of edges across partitions. In addition, the partitioned subgraphs cannot form *new* cycles; contexts waits for all input dependencies and therefore cycles across contexts cause deadlock. Nonetheless, the original graph might contain cycles representing loop carried dependencies, such as accumulation. For these cycles, SARA initialize the back edge of the cycle with dummy data to enable execution. Figure 4.13 shows examples of valid and invalid partitioning solutions. Figure 4.14 shows another example partitioning for a dataflow graph with cycle.

Community Detection The formulation of compute partitioning is similar to the community detection problem[**?**], which has similar objective. The major difference is that the later

often takes the number of output partitions as an input to the algorithm, whereas our problem partitions until all subgraphs satisfy all constraints. Moreover, community detection algorithms do not enforce the cycle constraints. Finally, the edge connectivity in community detection counts the number of edges across partitions, as supposed to broadcast edges.

Retiming Imbalanced data paths across partitions can cause pipeline stalls at runtime. To ensure full-throughput pipelining, SARA needs to insert retiming buffers along imbalanced data path across partitions. Retiming introduces new VUs in addition to the partitioned VUs, which attributes to the cost in Table 4.4's objective.

In the following sections, we present two algorithms to solve this problem: a traversal-based algorithm providing a decent solution with fast compile time, and a convex optimization-based algorithm with an optimum solution but long compile time.

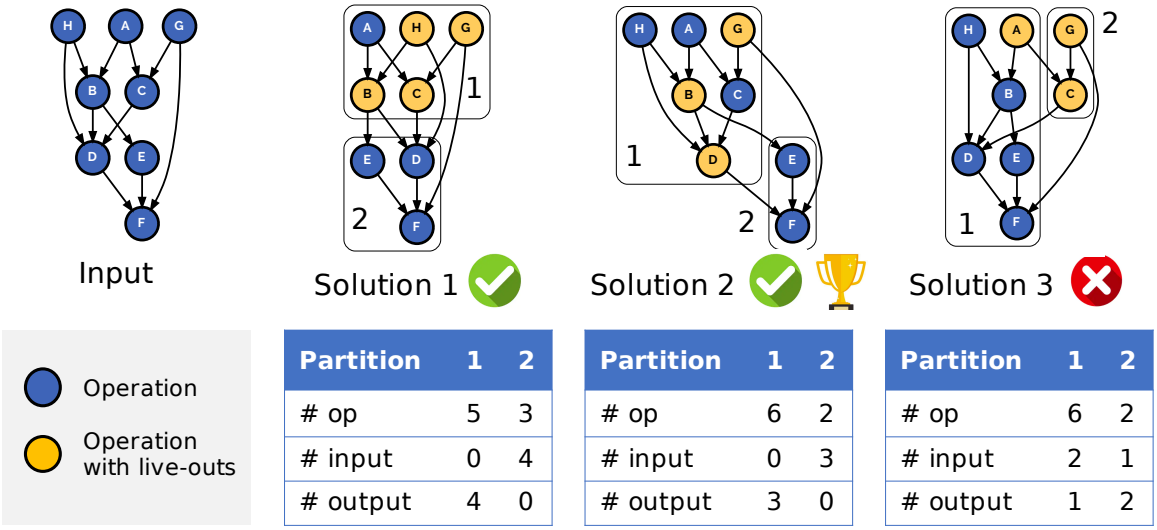
Traversal-based Solution

To address the cycle constraint, the traversal-based algorithm performs a topological sort of the dataflow graph. The topological traversal ignores the back edges in the graph. Staring from the beginning of the sorted list, the algorithm iteratively adds nodes into a partition until it fits no more nodes. The algorithm then repeat the process with a new partition. This approach guarantees that no cycle is introduced with $O(V + E)$ complexity, where V and E are the numbers of vertices and edges in the dataflow graph.

The partitioning result is a function of the traversal order. We experienced with depth-first search (DFS) and breadth-first search (BFS) with forward and backward dataflow traversal orders. For DFS, we re-sort the remaining list each time we start with a new partition.

Solver-based Solution

The convex optimization solution models the problem as a node to partition assignment problem. Table 4.6 gives our formulation. At a high-level, we use a boolean matrix B to keep track of the assignment. B has dimension equals to the number of nodes in the

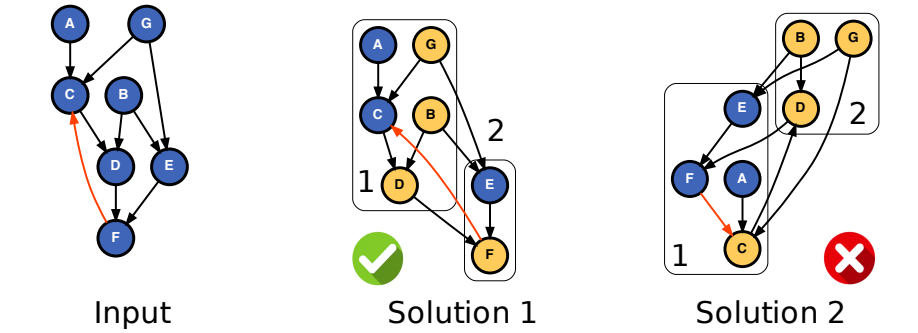


Partition	1	2
# op	5	3
# input	0	4
# output	4	0

Partition	1	2
# op	6	2
# input	0	3
# output	3	0

Partition	1	2
# op	6	2
# input	2	1
# output	1	2

Figure 4.13: Compute partitioning examples. Solution 1 and 2 are both valid partitioning. Solution 2 is better because it has less number of broadcast edges (3 as supposed to 4 in Solution 1) across partitions. Solution 3 is an example of illegal partition result due to the cycle between partition 1 and 2.



Partition	1	2
# op	5	2
# input	1	3
# output	3	1

Partition	1	2
# op	4	3
# input	3	1
# output	1	3

Figure 4.14: Compute partitioning examples with cycle in the dataflow graph. Solution 1 is valid because there is no cycle between partitions after removing the back edge in the original graph. Solution 2 is invalid because there is still cycle between partition 1 and 2 after removing the back edge.

dataflow graph by the number of partitions, where$B[i, j] = 1$ indicates node i is assigned to partition j. In Table 4.6, *partition assignment* restricts each node to have a single partition assignment. The *input and output arity constraints* show the formulations that limit the number of input and output for a subgraph. These are the two most challenging constraints as we need to identify broadcast edges across partitions. To address the cycle constraint, we introduce a delay vector d_n with size equivalent to the number of nodes. The delay vector encodes a schedule to execute each node and the values are selected by the solver. The *dependency constraint* enforces scheduling a node no earlier than its input dependencies, and no later than its output dependents. Since the operations within a partition has to be triggered atomically, there is another delay vector d_p for the partitions. The *delay consistency* enforces the schedule of the node equals to the schedule of its assigned partition. Finally, *constant validity* limits the range of values the delay values can be chosen from. In addition to enforcing the cycle constraint, these delay variables are also used to calculate where retiming is required and project the amount of introduced retiming VUs. Finally, to reduce the solving time, we use the traversal-based solution to warm start the assignment matrix B.

Comparison

TODO: Benchmark Table Figure 4.15 shows the comparison between the traversal-based and the solver-based solutions for both compute partitioning and global merging. Global merging is an global optimization merging small VUs into a large VU that can still fit in a PU. The merging algorithm is very similar to the compute partitioning algorithm, where nodes in the dataflow graph correspond to the VUs in a VUDFG. The merging problem also has a traversal-based and solver-based solution. Section 4.6 will discuss about merging in more details.

We used a commercial solver Gurobi [8] for the solver-based solutions. The evaluation is performed on a Intel Xeon E7-8890 CPU at 2.5GHz with 1TB DDR4 RAM. Gurobi is parallelized across 10 threads for each application. To speedup convergence, we configured Gurobi with 15% optimality gap, i.e. the solver is allowed to early stop after the solution

Name	Type	Description	Definition / Default
$\mathcal{N}$	Constant	Enumeration of nodes to partition, numbered $\{n_i\}_i$	-
N	Constant, $\mathbb{Z}_{\geq 0}$	Number of operations to partition	$N = \lvert\mathcal{N}\rvert$
P	Constant, $\mathbb{Z}_{\geq 0}$	Number of partitions to consider	N, or from heuristic
$\mathcal{E}$	Constant, $\{n_i \to n_j\}$	Directed edges representing dependence	-
B	Variable, $\{0,1\}^{N\times P}$	Boolean Partitioning Matrix	-
$\text{proj}_{\mathbf{B}}(\cdot)$	$\mathbb{Z}_{\geq 0} \to \mathbb{B}$	Function to convert a positive integer into a boolean	Supplemental Materials
$\text{and}(\cdot,\cdot)$	$\{0,1\} \times \{0,1\} \to \{0,1\}$	Boolean and of binary variables	Supplemental Materials
d_p	Variable, $\mathbb{Z}_{\geq 0}^P$	Vector of partition delays	-
d_n	Variable, $\mathbb{Z}_{\geq 0}^N$	Vector of node delays	-
$\text{dest}(n)$	$\mathcal{N} \to \mathcal{P}(\mathcal{N})$	The set of nodes which depend on n	$\{n' \lvert n' \in \mathcal{N}\ s.t.\ (n \to n') \in \mathcal{E}\}$
c_o	Constant, $\mathbb{Z}_{\geq 0}$	Maximum output arity of a partition	HW Spec
c_i	Constant, $\mathbb{Z}_{\geq 0}$	Maximum input arity of a partition	HW Spec
b_d	Constant, $\mathbb{Z}_{\geq 0}$	Maximum input buffer depth	HW Spec
K	Constant, $\mathbb{R}_+$	Very Large Constant, used for constraint activation	$P \times N$
α_d	Hyperparameter, $\mathbb{R}_+$	Retime merging probability multiplier	$\frac{1}{\max\{c_o,c_i\}}$
$\mathcal{C}_r$	$[\mathcal{N} \to \mathbb{R}_+, \mathbb{R}_+, [\mathbb{R}_+] \to \mathbb{R}_+]$	List of per-node values, limits, and reduction functions for reducible constraints	Supplemental Materials
F	$\{0,1\}^{N\times P}$	Feasibility matrix, whether a partition can support a node	HW Spec

Table 4.5: Names and definitions used in the solver-based algorithms.

Type	Description	Expression
Cost Function	Allocated Partitions	$\Sigma_i \text{proj}_{\mathbf{B}}(\Sigma_j B_{i,j})$
	Retiming Partitions	$\alpha_d \Sigma_{n_i \to n_j \in \mathcal{E}} \text{proj}_{\mathbf{B}}(\max\{d_n(j) - d_n(i) - b_d, 0\})$
Partition Constraint	Partition Assignment	$\forall n_i \in \mathcal{N} : \ \Sigma_j B_{i,j} = 1$
	Input Arity Constraint (vectorized)	$\Sigma_{n_i \in \mathcal{N}} \max\{\text{proj}_{\mathbf{B}}(\Sigma_{n_j \in \text{dest}(n_i)} B_{j,:}) - B_{i,:}, 0\} \leq c_i \times \vec{1}$
	Output Arity Constraint	$\forall p \in [0, P) :$ $\Sigma_{n_s \in \mathcal{N}} \text{and}(B_{s,p}, \text{proj}_{\mathbf{B}}(\max\{(\Sigma_{n_d \in \text{dest}(n_s)} B_{d,p}) - K \times B_{s,p}, 0\})) \leq c_o$
	Dependency Constraint	$\forall n_i \to n_j \in \mathcal{E} : \ d_n(i) + 1[p_i \neq p_j] \leq d_n(j)$
	Delay Consistency	$\forall n_i \in \mathcal{N} : \ d_n(i) \leq \min_j(d_p(j) + K - B_{i,j} \times K)$ $\forall n_i \in \mathcal{N} : \ d_n(i) \geq \max_j(d_p(j) + B_{i,j} \times K - K)$
	Constant Validity	$\forall n_i \in \mathcal{N} : \ d_n(i) \leq K$ $\forall i \in [0, P) : \ d_p(i) \leq K$
Merge Constraint	Feasibility Constraint	$\forall i, j \in [0, N) \times [0, P) : \ B_{i,j} \leq F_{i,j}$
	Reducible Constraints	$\forall j \in [0, P). \forall (c(\cdot), c_v, r(\cdot)) \in \mathcal{C} :$ $r([c(n_i) \times B_{i,j}]_{n_i \in \mathcal{N}}) \leq c_v$

Table 4.6: Solver formulation for partitioning. Expressions are presented using the Disciplined Convex Programming ruleset [1, 2]. Explanations for selected expressions can be found in the supplemental material.

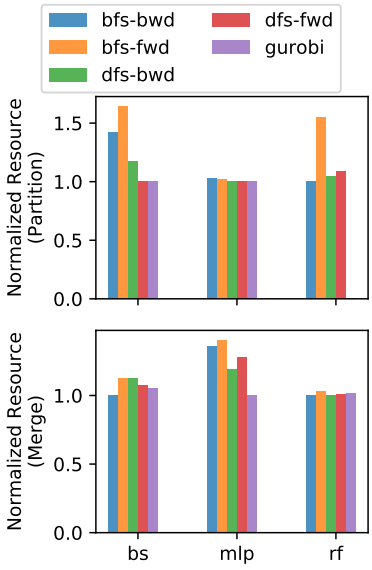
is more than 85% close to the optimum. The solving time increase dramatically as getting close to 100% optimum.

Figure 4.15 (a) shows the normalized resource in number of VUs after partitining and merging. We can see that Gurobi provides almost the best solution for all application when it can derive an answer in a reasonable amount of time. The missing solver bar in random forest (*rf*) partitioning is due to timeout after a few days. The traversal-based algorithms can sometimes match or even outperform the solver slightly. However, because the partitioning result is a function of the traversal order, each traversal order has adversarial cases, where they can be up to 1.7x worse in resource than the best possible solution. The forward (*fwd*) traversal order schedules nodes as earlier as possible, which reduces the number of external live variables; the backward traversal minimizes the number of internal live variables across partitions. The depth-first-search (DFS) traversal order minimizes the number of live variables between partitions, albeit producing more imbalanced paths between partitions. On the other hand, breath-first-search (BFS) produces more balanced partitioning with more live variables and partitions.

There are two common graph patterns in applications that require partitioning. The first is a data-flow graph from a large basic block, which contains a small set of external live-in and live-out variables and abundant of intermediate temporary variables. Such graphs typically have long-live variables that require retiming. TODO: continue

Figure 4.15 (b) and (c) shows the compile time for these algorithms. The single-threaded traversal-based algorithm runs in minutes, which is significantly faster than the parallelized solver that takes hours to days. In general, the solver runtime becomes quickly unscalable with a large amount of VBs. TODO: show solver time with increase number of VBs.

In a summary, the solver solution provides a guaranteed close-to-optimum solution at an expensive compile time. The solver-solution also treats the retiming and partitioning as a joint optimization, whereas these two problems are solved in two separate passes in the traversal-based solution. Moreover, the solver-based solution tends to produce better result for an architecture with tight I/O bound and relax stage constraint. The traversal-based solutions, on the other hand, can produce a decent solution in a short amount of



(a) Resource Comparison

Apps	bfs-bwd	bfs-fwd	dfs-bwd	dfs-fwd	gurobi
bs	3s	3s	2s	2s	2h4m
mlp	<1s	<1s	<1s	<1s	4s
rf	1m37s	1m7s	29s	27s	-

(b) Compile time for splting

Apps	bfs-bwd	bfs-fwd	dfs-bwd	dfs-fwd	gurobi
bs	1s	2s	5s	3s	1m4s
mlp	5s	5s	6s	11s	31m16s
rf	11s	10s	2m7s	1m0s	14h27m

(c) Compile time for merging

Figure 4.15: Partitioning and merging algorithm comparisons. (a) shows the normalized resource usage between different algorithms (the lower the better). (b) and (c) shows the compile time of each algorithm.

time. However, the solution is prune to adversarial cases and potentially perform badly for unseen graph structure. In practice, we can combine the two approaches and invoke the expensive solver only when the traversal-based solution is insufficient. The quality of partitioning can be easily estimated by the resource utilization in each partition.

4.5.2 Memory Partitioning

The memory pruner addresses VUs with virtual on-chip scratchpad memories exceeding the physical limits in capacity or number of banks in a PU. Memories in the input graph can have arbitrary size and number of virtual banks. The PUs, on the other hand, contains a small number of fix-sized 1-D scratchpad banks.

To partition the virtual memory, SARA shards the large virtual memory into multiple memory partitioned VUs, and assign each partition with a subset of the virtual banks. Each

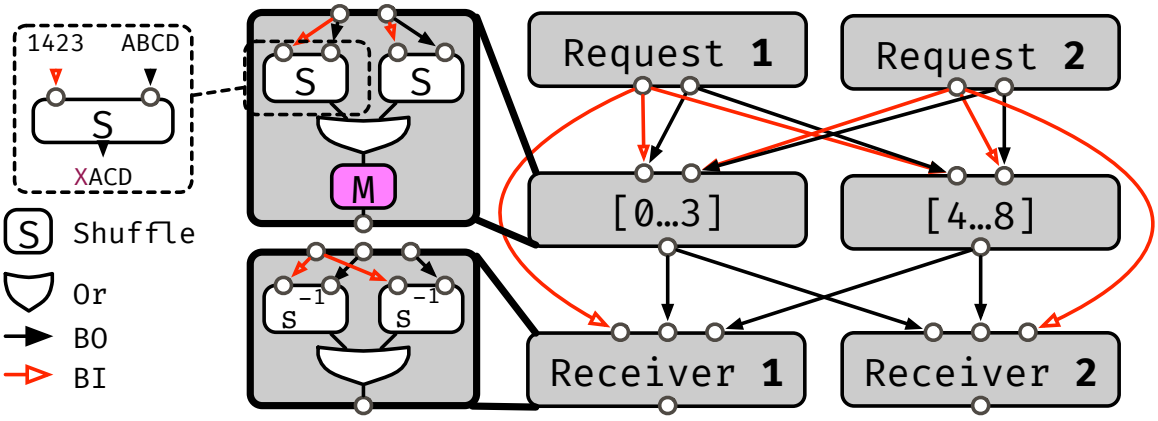


Figure 4.16: An example of splitting a memory to serve parallel requesters.

accessor provides a bank ID (BI) that selects which banks to access and a bank offset (BO) that specifies the address within the bank. BI and BO can be vectorized. SARA can use multiple banks within a PU to form a larger virtual bank. However, if a virtual bank in VU exceeds the total capacity of all physical banks within a PU, SARA further partitions the large virtual banks into multiple sub-banks, such that each sub-bank can fit into the aggregated capacity of a PU. To do so, SARA injects additional calculation to derive the sub-bank ID and sub-bank offset from the BO, and flattens the sub-bank ID with the previous BI to form the new BI and the new BO.

SARA then set ups the crossbar data path between the parallel producers, memory partitions, and parallel consumers. As discussed in Section 4.4.3, each accessor is split into a requester context and receiver context. For each memory partition, SARA uses an context to merge the requests from all parallel requesters, Figure 4.16. The merge context uses a special shuffle operator that shuffles the BO vector from the order in BI to the order aligned with banks in its partition. If a bank in the partition is not accessed in BI, the output of the shuffle is marked as invalid. We assume the capacity of each physical bank is much smaller than 2^{31}. So we use the first bit of the 32-bit BO to indicate invalid accesses (0 is invalid), which is also used to explicitly disabled access from the program. Next, the merger context uses a tree of bit-wise OR operators to combine all bank-aligned BOs, and send the combined request vector to its partition. The requests can be trivially ORed because static

banking (**??**) guarantees that no two requesters access the same bank in the same cycle. Next, the memory partitions broadcast the respond to all receiver contexts. A receiver takes response from all partitions, using the same shuffle operators to align each response back to the requested order in BI, and uses another OR tree to merge the response. The BI is forwarded from the requester to the receiver for the reverse shuffling.

The alternative approach is to reverse the respond to access ordering within the memory partitions before sending them to the requester. This approach does not scale with network bandwidth, as the memory partitions need to send the receiver number of distinct outputs. (The number of receivers is a function of the parallelization factor.) As a result, the amount of output bandwidth at the memory partition limits how much the program can be parallelized, which causes underutilization of the accelerators. In our scheme, each partition sends a single broadcast to all receivers, which is efficiently handled by the network.

The request trees for memory partition and receiver can have high fan-in, which can be partitioned into a tree of VU during the compute partitioning phase in Section 4.5.1.

4.6 Optimizations

SARA performs many of the standard compiler optimizations, such as Dead Code Elimination and Constant Propagation. Some of them, however, plays a much more important rule for reconfigurable accelerator because they have direct impact on the resource usage. Other optimizations can be counter-intuitive, as they introduce redundant computation that reduces resource without necessarily impacting performance. In this discussion, we focus our primary objective on performance. Resource is an indirect objective as resource reduction enable larger parallelization factors, , which in turn improves performance.

Memory strength reduction (msr). Like traditional strength reduction on arithmetics, SARA replaces expensive on-chip memories with cheaper memories whenever possible. For example, SARA replaces a scratchpad with constant address in all accesses to a unindexable memory, such as a FIFO. This commonly happens when producer and consumer loops of the memory are fully unrolled.

Route-Through Elimination (rtelm). For patterns where the content of a non-indexable memory (M1) is read and written to another memory (M2), SARA eliminates the intermediate access if the read of M1 and the write of M2 operates in lock-step.

Retiming with scratchpad (retime-mem). By default, SARA uses PB input buffers for retiming purpose. This option enables SARA to use scratchpad memory for retiming that requires large buffer depth.

Crossbar datapath elimination (xbar-elm). Although, crossbars between accessors and memory partitions (Section 4.5.2) are very expensive in the general case, the BI sometimes can be statically resolvable with certain combinations of parallelization on memory accesses. When BI is a constant, SARA can use this information to intelligently assign virtual banks to partitions that reduce the crossbar data path to a partial or a point-to-point connection.

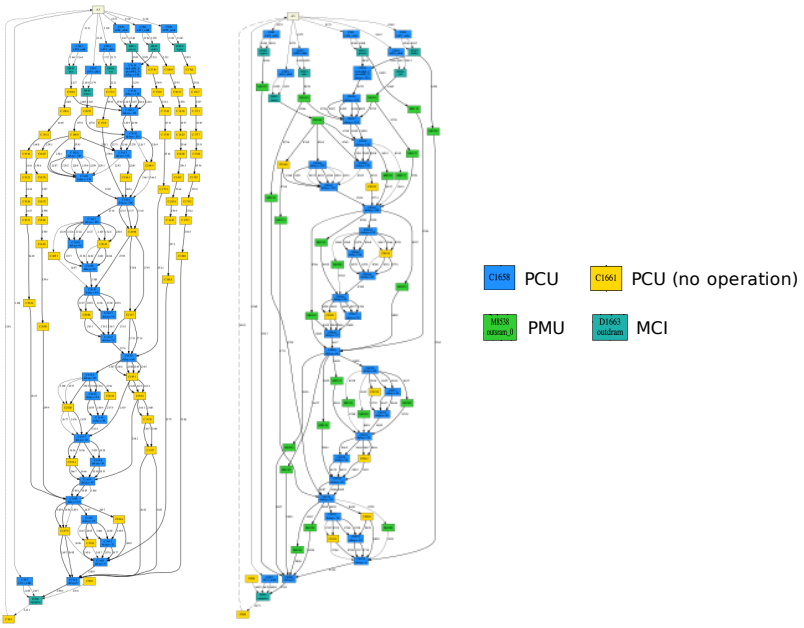
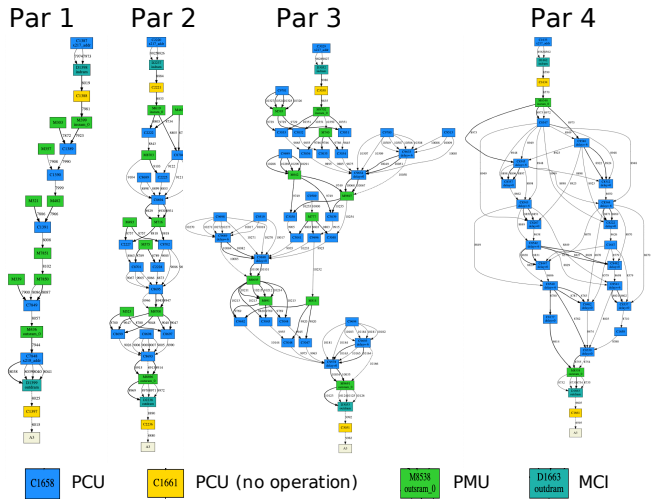


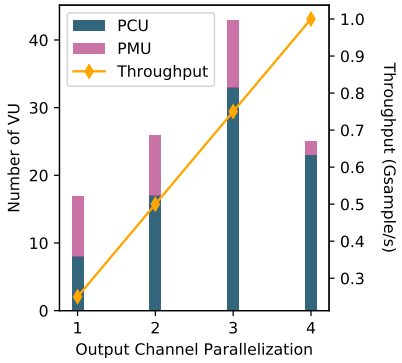
Figure 4.17: Left: retiming with input buffer only. Right: retiming with either input buffer or scratchpad.

Read request duplication (dupra). During memory partitioning (Section 4.5.2), instead of forwarding BI from the requester to receiver, SARA can also duplicate the BI with local state on receiver side, which eliminates the unbalanced data path at the cost of extra computation.

Global Merging (merge). After all VBs satisfy the hardware constraint, we perform a global optimization to compact small VBs into larger VBs. Merging has very similar problem statement as compute partitioning (Section 4.5.1) except with more constraints. The traversal-based algorithm requires a reference cost for PB to check if the merged VB still satisfy the hardware constraint. At each step of merging, we take the union of the domains of VBs within the current partition, and intersect with the domain of the merging VB. The caveat is that even with a non-empty intersection, the bipartite graph might not have a possible assignment, as merged VB might fit in a larger PB with insufficient quantity. Therefore, we perform a heuristic checking on feasibility of the bipartite assignment at each step of merging. The solver-based solution combines partition assignment with PB



(a) VUDFGs with different parallelization factors



(b) Throughput and resource as a function of parallelization factor

Figure 4.18: MLP case study

type assignment as a joint problem. The output of merging gives both partition assignment as well as a PB type assignment, which eliminates the risk of un-mappable bipartite graph due to merging in the traversal-based solution.

Memory Localization We perform another specialization on non-indexable memories (registers or FIFOs), whose all accesses have no explicit read enables. Instead of treating them as shared memories, SARA duplicates and maps them to local input buffers in all receivers, no longer requiring tokens. The sender actor pushes to the network when the token is supposed to be sent, and the receiver dequeues one element from the input buffer when the token is supposed to be consumed. This dramatically reduces the synchronization complexity of non-indexable memory in the common case.

Reverse Loop Invariant Hoisting A common loop optimizations is to move loop invariant expressions outside of the loop body to reduce computation. This optimizations, however, might introduce more basic blocks in the program. For Plasticine, number of basic blocks have a strong correlation with number of contexts and physical units. Figure 4.20 shows an example where moving instructions into the loop body reduces number compute

```
1 mem1 = rand(N)
2 mem2 = rand(N)
3 # Block 1
4 c = a + b
5 for i in range(N):
6     # Block 2
7     mem2[i] += mem[i] * c
```

(a) Input program

```
1 mem1 = rand(N)
2 mem2 = rand(N)
3 for i in range(N):
4     # Block 2
5     c = a + b
6     mem2[i] += mem[i] * c
```

(b) Reverse Loop Invariant Hoisting

Figure 4.19: The original program requires at least two contexts to execute Block 1 and Block 2. By moving the invariant instruction `c = a + b` into the loop body, (b) only needs a single context instead. Because instructions within `Block 2` are pipelined across loop iterations, adding instructions in the loop body introduce minimum performance impact. This transformation is beneficial until `Block 2` exceeds six operations, at which point both version consume the same amount of resources.

units. Because instructions within basic blocks and basic blocks themselves are pipelined, doing so does not have performance impact on the application. Currently, we rely on the user to perform manually perform this optimization.

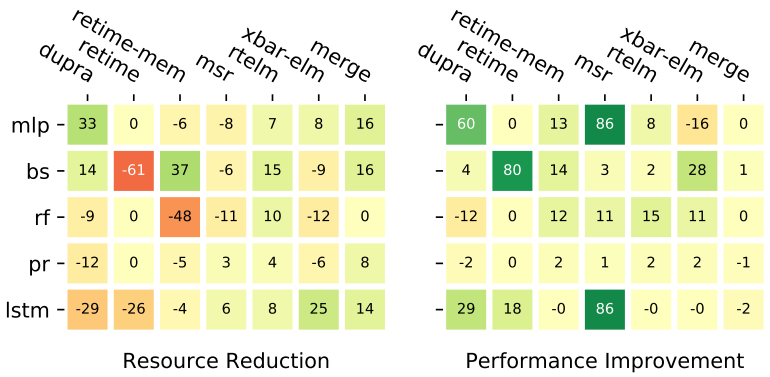


Figure 4.20: Optimization Effectiveness. Percentage resource reduction or performance improvement from each optimization. The heat map shows the maximum differences when turning on/off the optimization, while keeping other optimizations the same. For a single application, the improvement is taking the geometric mean across different application parameters.

4.7 Register Allocation

4.8 Debugging and Instrumentation Support

Deadlock in Streaming Reconfigurable Architecture

Debugging Support and Performance Instrumentation

Chapter 5

Related Work

5.1 Network

Multiple decades of research have resulted in a rich body of literature, both in CGRAs [**?**, **?**] and on-chip networks [**?**]. We discuss relevant prior work under the following categories:

5.1.1 Tiled Processor Interconnects

Architectures such as Raw [**?**] and Tile [**?**] use scalar operand networks [**?**], which combine static and dynamic networks. Raw has one static and two dynamic interconnects: the static interconnect is used to route normal operand traffic, one dynamic network is used to route runtime-dependent values which could not be routed on the static network, and the second dynamic network is used for cache misses and other exceptions. Deadlock avoidance is guaranteed only in the second dynamic network, which is used to recover from deadlocks in the first dynamic network. However, as described in Section **??**, wider buses and larger flit sizes create scalability issues with two dynamic networks, including higher area and power. In addition, our static VC allocation scheme ensures deadlock freedom in our single dynamic network, obviating the need for deadlock recovery. The dynamic Raw network also does not preserve operand ordering, requiring an operand reordering mechanism at every tile.

TRIPS [?] is a tiled dataflow architecture with dynamic execution. TRIPS does not have a static interconnect, but contains two dynamic networks [?]: an operand network to route operands between tiles, and an on-chip network to communicate with cache banks. Wavescalar [?] is another tiled dataflow architecture with four levels of hierarchy, connected by dynamic interconnects that vary in topology and bandwidth at each level. The Polymorphic Pipeline Array [?] is a tiled architecture built to target mobile multimedia applications. While compute resources are either statically or dynamically provisioned via hardware virtualization support, communication uses a dynamic scalar operand network.

5.1.2 CGRA Interconnects

Many previously proposed CGRAs use a word-level static interconnect, which has better compute density than bit-based routing [?]. CGRAs such as HRL [?], DySER [?], and Elastic CGRAs [?] commonly employ two static interconnects: a word-level interconnect to route data and a bit-level interconnect to route control signals. Several works have also proposed a statically scheduled interconnect [?, ?, ?] using a modulo schedule. While this approach is effective for inner loops with predictable latencies and fixed initiation intervals, variable latency operations and hierarchical loop nests add scheduling complexity that prevents a single modulo schedule. HyCube [?] has a similar statically scheduled network, with the ability to bypass intermediate switches in the same cycle. This allows operands to travel multiple hops in a single cycle, but creates long wires and combinational paths and adversely affects the clock period and scalability.

5.1.3 Design Space Studies

Several prior studies focus on tradeoffs with various network topologies, but do not characterize or quantify the role of dynamism in interconnects. The Raw design space study [?] uses an analytical model for applications as well as architectural resources to perform a sensitivity analysis of compute and memory resources focused on area, power, and performance, without varying the interconnect. The ADRES design space study [?] focuses

on area and energy tradeoffs with different network topologies with the ADRES [**?**] architecture, where all topologies use a fully static interconnect. KressArray Xplorer [**?**] similarly explores topology tradeoffs with the KressArray [**?**] architecture. Other studies explore topologies for mesh-based CGRAs [**?**] and more general CGRAs supporting resource sharing [**?**]. Other tools like Sunmap [**?**] allow end users to construct and explore various topologies.

5.1.4 Compiler Driven NoCs

Other prior works have used compiler techniques to optimize various facets of NoCs. Some studies have explored statically allocating virtual channels [**?**, **?**] to multiple concurrent flows to mitigate head-of-line blocking. These studies propose an approach to derive deadlock-free allocations based on the turn model [**?**]. While our approach also statically allocates VCs, our method to guarantee deadlock freedom differs from the aforementioned study as it does not rely on the turn model. Ozturk et al. [**?**] propose a scheme to increase the reliability of NoCs for chip multiprocessors by sending packets over multiple links. Their approach uses integer linear programming to balance the total number of links activated (an energy-based metric) against the amount of packet duplication (reliability). Ababei et al. [**?**] use a static placement algorithm and an estimate of reliability to attempt to guide placement decisions for NoCs. Kapre et al. [**?**] develop a workflow to map applications to CGRAs using several transformations, including efficient multicast routing and node splitting, but do not consider optimizations such as non-minimal routing.

5.2 Compiler

Streaming Dataflow IRs Although many works claim to emit efficient and information-rich dataflow IRs for the downstream compilers, very few of them can capture the high-level parallel patterns and implementation details that are critical to RDA mappings. For example, TensorFlow [**?**] emits dataflow IR composed of tensor operations. However, its IR lacks information on the parallel patterns within these operations. In contrast, most

of the streaming languages [**?**, **?**, **?**] are not able to extract nested loop-level parallelism from modern data-intensive applications. For example, StreamIt [**?**], a language tailored for streaming computing, also adopts distributed control as in SARA. However, it lacks the necessary language features to describe deeply and irregularly nested loops that are common in modern data-intensive applications.

Hardware Architectures Spatial reconfigurable accelerators (*e.g.*, Dyser [**?**] and Tartan [**?**]) have only one-level of hierarchy. Hence, such accelerators' performance can be bottlenecked by their limited interconnect bandwidth and power budget. Sparse Processing Unit (SPU) [**?**] can sustain higher interconnect bandwidth by introducing on-chip hierarchy; however, it lacks support for polyhedral memory banking [**?**], a pivotal optimization to achieve massive parallel accesses to on-chip memory. Plasticine [5] provides us with the desired architecture features; however, its compiler lacks the necessary components to support efficient streaming execution. Given that Plasticine resembles many key features of the RDA model, we target Plasticine with SARA.

Spatial Compilers Most previous works [**?**, **?**] only consider allocating resources at the same level. SARA takes a more general assumption by co-allocating resources at multiple levels of an accelerator's hierarchy.

The Plasticine compiler [5] is similar to SARA that it also uses a token-based control protocol. However, it performs worse than SARA due to the following reasons. First, the Plasticine compiler allocates VBs for every level of Spatial's (a high-level language) control hierarchy. The communication between parent and child controllers lead to both communication hotspots around the parent, and bubbles before entering a steady-state of the loop iterations. Second, the Plasticine compiler assigns a single memory PB for each logical memory in the Spatial program. Hence, it could not handle the case where a logical memory exceeds the capacity or bank limits of the physical PBs. Third, the Plasticine compiler only supports polyhedral memory partitioning at the first dimension of the on-chip

memory. Hence, its applicability to data-intensive applications with high-dimension tensor algebra is questionable. Last, compared to SARA's separate allocation and assignment phases described in Section 4.4 and Section 4.5, the Plasticine compiler allocates one VB for a specific type of PB and underutilizes resources within PBs.

Chapter 6

Future Work

Chapter 7

Conclusions

18 For I testify unto every man that heareth the words of the prophecy of this book, If any man shall add unto these things, God shall add unto him the plagues that are written in this book:

19 And if any man shall take away from the words of the book of this prophecy, God shall take away his part out of the book of life, and out of the holy city, and from the things which are written in this book.

Appendix A

Appendix

A.1 Context Programming Restrictino

We use the imperative context configuration for ease of understanding, but it is important to realize not all program expressible at this abstraction can be executed by the PCU.

```
with Context() as ctx:
    bufferA = VecInSteram()
    bufferB = VecInStream()
    bufferN = ScalInStream()
    bufferAcc = ScalOutStream()

    for i in range(0, N, 16)
        a = bufferA.deq()
        b = bufferA.deq()
        # vectorized multiply
        prod = a * b
        # produce a scalar ouptut
        r = reduce(prod, lambda a,b: a+b)
        # scalar accumlation in PR
        accum += r
    # sends bufferAcc every N/16 cycles
    bufferAcc.enq(accum)
```

(a) Declarative configuration

```
with Context() as ctx:
    bufferA = VecInSteram()
    bufferB = ScalInSteram()
    bufferC = VecInSteram()
    outputD = VecOutSteram()

    b = bufferB.deq()
    for i in range(0, 3, 1)
        c = bufferC.deq()
        for j in range(0, 3, 1)
            a = bufferA.deq()
            expr = a * b + c
            outputD.enq(data=expr)
```

(b) Imperative configuration

Bibliography

[1] M. Grant, *Disciplined Convex Programming*. Phd. thesis, Stanford University, 2014.

[2] Y. Y. Michael Grant, Steven Boyd, "Disciplined convex programming," 2019.

[3] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), pp. 296–311, ACM, 2018.

[4] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 115–127, June 2016.

[5] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 389–402, ACM, 2017.

[6] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pp. 1–14, IEEE Press, 2018.

[7] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pp. 1–12, IEEE, 2017.

[8] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2019.