SCALING A RECONFIGURABLE DATAFLOW ACCELERATOR

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
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 (WIP)

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 developed 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 work, we focus on two aspects of the software-hardware codesign that impact the usability 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 programmability challenge, we 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. To address the performance challenges, we 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 (WIP)

1.1 The Rising of Hardware Acceleration

With the end of Dennard Scaling [1], 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 [2]. 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[3].

For this reason, many recent efforts are spent on leveraging application domain knowledge in hardware design to enable continued performance scaling while meeting the power budget[4]. Examples include widely adopted General-Purpose Graphics Processing Units (GPGPUs) in the deep-learning domain and machine learning (ML) accelerators, such as Tensor Processing Units [5] and EIE [6], providing orders of magnitude acceleration over a CPU. However, massive threads in GPU and the highly specialized datapath in ML accelerators often cause severe underutilization of the hardware due to variation in application and data characteristics[7].

Reconfigurable spatial architectures overcome this limitation by changing its datapath based on applications' needs. Applications are configured at the circuit-level without dynamic instruction fetching and decoding, hence improving energy-efficiency. In addition to instruction, data, and task-level parallelism explored by processor architectures, spatial architectures also explore instruction and task-level pipelining that further increase the compute throughput. Pipelining at various granularity enables spatial architectures to achieve high throughput on program phases without massive parallel workload, and allocate resource proportional to compute intensities of program phases.

One example of a reconfigurable spatial architecture is Field Programmable Gate Arrays (FPGAs) that support fine-grain, bit-level reconfiguration with a soft logic fabric [8]. The flexible interconnect and lookup table-based logic gate can be configured to implement arbitrary datapath. FPGAs are used to deploy services commercially [9, 10, 11] and can be rented on the AWS F1 cloud [12]. Although around for a long time, FPGAs are not broadly used on high-level applications due to their low-level programming interface and long compilation time. Fine tuning applications on FPGAs requires expertise in digital design knowledge and takes a long development cycle, which hinders their accessibility to general software programmers. As an application-level accelerator, FPGAs also suffers from overhead incurred by fine-grained reconfigurability; studies have shown that an FPGA can have over have hampered widespread adoption for several years [13, 14, 15, 16].

Lately, Reconfigurable Dataflow Accelerators (RDAs) [17, 18] are emerging as a new class of spatial accelerators that retain the desired level of flexibility and energy efficiency without the fine-grained reconfigurability overhead. RDA provides high resource density and compute throughput with a hierarchical streaming network, operating at a fixed and high clock frequency at large chip sizes. Recent studies [17] have demonstrated a promising acceleration of dense, sparse, and streaming applications using RDAs.

Spatially reconfigurable architectures are programmable, energy efficient application accelerators offering the flexibility of software and the efficiency of hardware. Architectures such as Field Programmable Gate Arrays (FPGAs) achieve energy efficiency by providing statically reconfigurable compute elements and on-chip memories in a bit-level programmable interconnect; this interconnect can be configured to implement arbitrary datapaths. FPGAs are used to deploy services commercially [9, 10, 11] and can be rented on the AWS F1 cloud [12]. However, FPGAs suffer from overhead incurred by fine-grained reconfigurability; their long compile times and relatively low compute density have hampered widespread adoption for several years [13, 14, 15, 16]. Therefore, recent spatial architectures use increasingly coarse-grained building blocks, such as ALUs, register files, and memory controllers, distributed in a programmable, word-level static interconnect. Several of these Coarse-Grained Reconfigurable Arrays (CGRAs) have recently been proposed [19, 20, 21, 22, 23, 24, 18, 25, 17].

1.2 The Need of Flexible Interconnects

CGRAs need the right amount of interconnect flexibility to achieve high resource utilization; an inflexible interconnect constrains the space of valid application mappings and hinders resource utilization. Furthermore, in the quest to increase compute density, CGRA data paths now contain increasingly coarse-grained processing blocks such as pipelined, vectorized functional units [17, 22, 26]. These data paths typically have a vector width of 8–16x [17], which necessitates coarser communication and higher on-chip interconnect bandwidth to avoid creating performance bottlenecks. Although many hardware accelerators with large, vectorized data paths have fixed local networks [27], there is a need for more flexible global networks to adapt to future applications. Consequently, interconnect design for these CGRAs involves achieving a balance between the often conflicting requirements of high bandwidth and high flexibility.

1.3 The Gap between High-Level DSLs and Dataflow Accelerators

Recent years has seen a raise in interests in researching in hardware accelerators, primarily motivated by AI applications. On one hand, huge success of adoption of AI in various domain has motivated various machine learning specialized hardware in the architecture community. On the other hand, research in machine learning algorithms has been expedited by the increasing powerful hardware accelerators, enabling the next-level algorithms that was not feasible in just decades ago. Research in software infrastructure for these accelerators, however, are just emerging. Unlike CPUs, these accelerators do not support a standard ISAs, alleviating the overhead from layers of abstractions and the burden of backward compatibility. Nonetheless, lack of a common abstraction makes it very hard to sharing compiler infrastructure across accelerators.

1.4 Contribution

1.5 Outline

Chapter 2

Background (WIP)

2.1 Execution Schedules of Reconfigurable Architectures

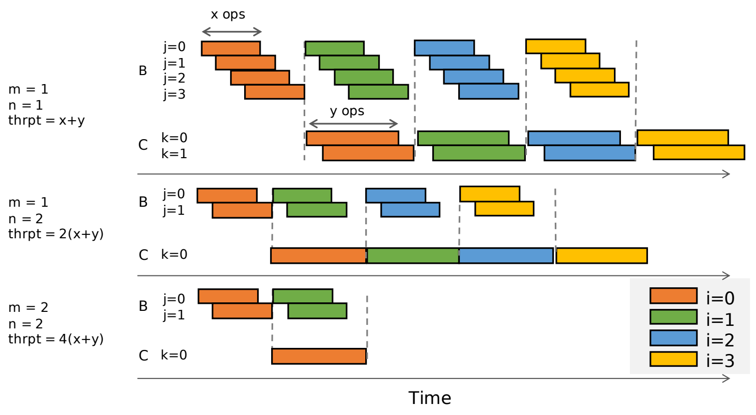
The key advantage of reconfigurable spatial accelerators, compared to processor-based architectures, is the ability to explore multiple levels of pipeline parallelism. In traditional Von Neumann architectures [28], like CPUs and GPUs, a computer consists of a processing unit that performs computation, 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 inherently assumes that instructions with in a program are executed in time, maximizing the flexibility to context switching between different workloads dynamically.

Reconfigurable accelerators are a direct violation of the von Neumann execution model; instructions are statically imbedded in the datapath and executed in space as supposed to in time. One of the disadvantage of reconfigurable hardware is paying the resource cost for infrequently executed instructions, making it unsuitable for control-heavy workloads that traditional processors are efficient at. On the other hand, RDAs are particularly competitive in providing high-throughput, low-latency, and energy-efficiency acceleration for data-analytical workloads. Data-analytical workloads encompass a wide domains of applications, including image processing, recognition, machine translation, digital signal processing, network processing, etc. These applications exhibits a rich amount of data-level parallelism with relatively static control flow.

Figure 2.1 shows an example of hierarchical parallelism and pipelining exploit by a spatial architecture. The overall compute throughput of a parallelized and pipelined program is the product

```
mem = array(dims=[100])
A: for i in range(A,par=m):
    B: for j in range(B,vec=n):
        # a basic block with x
        # operations
        waddr, wdata = ...
        mem(waddr) = wdata # W1
        ...
    C: for k in range(C,vec=n):
        # a basic block with y
        # operations
        raddr = ...
        rdata = mem(raddr) # R1
        ...
```

(a)



(b)

Figure 2.1: Hierarchical pipelining and parallelization in spatial architecture. (a) illustrates the runtime and throughput of a hierarchically 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 pipelined across the outer loop iterations. Exploiting multiple levels of pipeline parallelism gives a total throughput of $x + y$ operations per cycle, where x and y are number of operations in the basic blocks. (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 further increases the throughput to $(x+y)mn$.

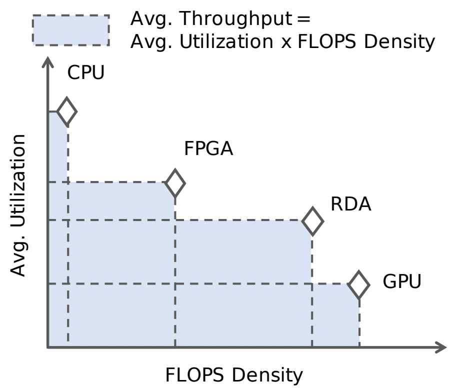


Figure 2.2: Average utilization vs. peak compute density tradeoff among different architectures.

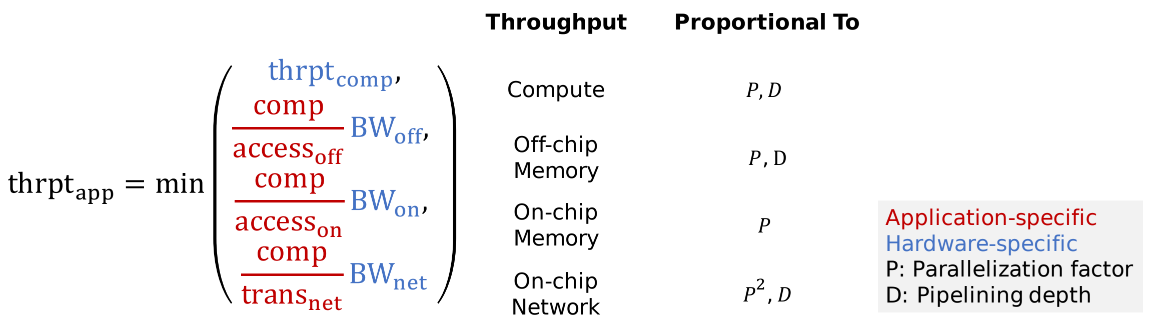

Throughput
Proportional To
$\text{thrpt}_{\text{app}} = \min\begin{pmatrix} \text{thrpt}_{\text{comp}}, \\ \frac{\text{comp}}{\text{access}_{\text{off}}}\text{BW}_{\text{off}}, \\ \frac{\text{comp}}{\text{access}_{\text{on}}}\text{BW}_{\text{on}}, \\ \frac{\text{comp}}{\text{trans}_{\text{net}}}\text{BW}_{\text{net}} \end{pmatrix}$
Compute
P, D
Off-chip Memory
P, D
On-chip Memory
P
On-chip Network
P^2, D
Application-specific
Hardware-specific
P: Parallelization factor
D: Pipelining depth

Figure 2.3: High-level performance model of spatial architectures

of the total parallelization factors and pipelining depth. By exploring multiple dimensions of concurrency in the program, spatial architecture is more likely to achieve a good compute throughput for a wide range of applications. For applications that are expensive to parallelize due to irregular access patterns, spatial architectures can increase on the pipelining dimension; for application with embarrassingly parallel workloads, spatial architecture can budget most resource on increasing parallelism.

Another benefit of pipelined execution is easier to achieve good memory performance. Data accessed by different stage of the pipelines are stored in discrete scratchpads instead of a shared cache; improving the effective on-chip bandwidth and capacity. Using explicitly managed scratchpad also tends to improve locality and eliminate cache performance issues, such thrashing. Across kernels, pipelined execution reduces the amount of off-chip accesses for intermediate data. SIMT architectures, like GPUs, relying on high-bandwidth DRAM technology, such has HBM, to sustain the compute throughput of massively parallelized threads. While providing over 10x more bandwidth than traditional DDR technologies, HBM is very limited in capacity, around 16GB as supposed to on the orders of TB for DDR. As a result, the limited off-chip capacity often restricts the type of applications that GPUs can support.

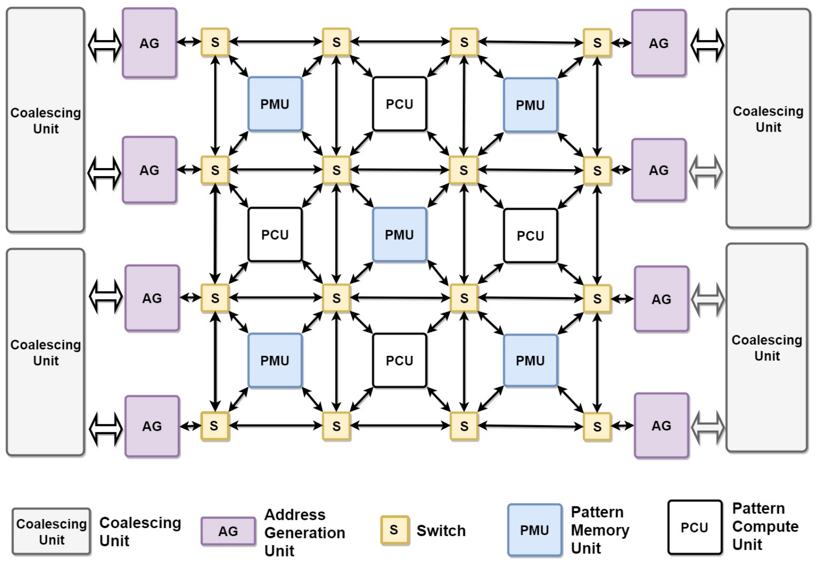


Figure 2.4: Plasticine chip-level architectural diagram

2.2 The Plasticine Architecture and its Programming Interface

Plasticine is a tile-based reconfigurable dataflow accelerator designed for a wide range of data-intensive workloads. With an area footprint of $113mm^2$ at 28-nm process, Plasticine packs 12.3 TFLOPS of compute throughput and 16 MB of on-chip memory. Without bit-level reconfiguration overhead, Plasticine runs a higher clock frequency than most FPGAs, running at 1GHz with a thermal design power at 49W. Previous work has shown a up to 77X performance-per-watt improvement from Plasticine over a Stratix V FPGA [17]. Figure 2.4 shows the chip-level design of Plasticine.

At high-level, Plasticine contains an array of resource tiles connected by a global interconnect. The mesh network is statically configured, providing a guaranteed in-order transmission of packet streams between any tiles. The network comes with three granularity: a 512-bit vector data bus, a 32-bit scalar data bus, and a single-bit control bus. The data networks contain a single valid bit traveled along with the data; the control network has only the valid bit without a payload, transmitting control pulses across tiles. There are three types of configurable units: the pattern compute units (PCUs) perform the most heavy lifting computation on Plasticine; the pattern memory units (PMUs) contain all distributed scratchpad on-chip; and the DRAM Address Generation Unit (AG) that generates DRAM requests going to the off-chip memory.

Like FPGAs, Plasticine can also support multiple-level of parallelization and pipelining explained in the previous section.

For the rest of this section, we will explain the native programming interfaces of the three configurable units.

Pattern Compute Unit (PCU)

As the major compute workhorse 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. The 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. 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 2.5(a) shows an example of a 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, 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 dataflow architectures, the dataflow engine in Plasticine is more flexible in that it allows dynamic enqueue and dequeue window for its input and output streams. *This feature enables Plasticine to support complex control hierarchy, such as non-perfectly nested loops and branch statements, across contexts even though individual contexts can only execute instructions that are control-free*.

Figure 2.5(c) 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 realize the instructions in the imperative-style configurations are not executed in time, but rather in space. There is a one-to-one translation from the declarative-style configuration to the imperative-style, but not in a reverse way. Instructions in the contexts must belongs to a single basic block, and the loops need to be perfectly nested (only accesses to streams can occur in the outer loops).

```
# 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", b, c)
# Operands are PRs from the previous stage
stage(1,"add",dst="PR2",oprd=["PR0","PR1"])

# forwarding PR2 of stage 1 to stage 2
forward(stage=2,dst="PR2",src="PR2")
forward(stage=3,dst="PR2",src="PR2")
forward(stage=4,dst="PR2",src="PR2")
forward(stage=5,dst="PR2",src="PR2")
forward(stage=0,dst="PR3",src=j.valid())
forward(stage=1,dst="PR3",src="PR3")
forward(stage=2,dst="PR3",src="PR3")
forward(stage=3,dst="PR3",src="PR3")
forward(stage=4,dst="PR3",src="PR3")
forward(stage=5,dst="PR3",src="PR3")
outputD.enq(data="PR2", when="PR3")
```

(a) Declarative configuration

```
bufferA = VecSteram()
bufferB = ScalarStream()
bufferC = VecSteram()
outputD = VecSteram()

with Context() as ctx:
    b = bufferB.deq()
    i = Counter(0,3,1)
    j = Counter(0,3,1)
    ctx.chain(i,j)
    b = bufferB.deq(when=i.done())
    c = bufferC.deq(when=j.done())
    a = bufferA.deq(when=j.valid())

    expr = a * b + c
    outputD.enq(data=expr, when=j.valid())
```

(b) Declarative configuration with automatic register allocation

```
bufferA = VecSteram()
bufferB = ScalarStream()
bufferC = VecSteram()
outputD = VecSteram()

with Context() as ctx:
    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(expr)
```

(c) Equivalent imperative program

Figure 2.5: 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. Each stage contains multiple pipeline registers (PRs) that can propogate results across stages. A stage can read PRs from the previous stage and write to PRs in its current stage. Only the first stage can read streams and counter values and the last stage can write streams. Other stages need to propogate the required values through PRs. (b) shows a simplified configuration where registers are implicitly allocated. 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. (c) shows the effective execution achieved in an imperative program. In (b), 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. In (b) and (c), we move the definitions of streams outside of the context, so they can be written by other contexts. Each stream can have exactly one writer. If a stream has more than one reader, effectively the writer broadcasts the result to both input buffers of the receiver contexts.

There is no global scheduler to orchestrate the execution order among contexts—the execution is purely streaming and dataflow driven. The only way to order the execution of two independent contexts is to introduce a control **token** between two contexts acting like a dummy data-dependency. This restriction eliminates the possible long-traveling wires and communication hot spots 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 the 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 the total resources consumed by all contexts do not exceed the 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 contexts such that the memory access order is consistent with the program order from a high-level imperative programming language.

Unlike most accelerators at this scale, Plasticine does not have any shared global on-chip memory. This design dramatically improves 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 a 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 unpredictable latency on both control and data paths. One of the major contributions of SARA is to hide these synchronization burdens 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 (AG) specialized in generating off-chip requests. Like compute pipeline in PMUs, the pipeline in AG is also non-vectorized with integer arithmetics. Each AG can generate 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 responses within each stream and no ordering guaranteed across streams. In streaming pipelined execution, the program can use multiple streams for DRAM accesses appeared in different locations of the program. To provide off-chip memory consistency, SARA allocates synchronizations across these streams to preserve memory order expected by the program.

2.3 The High-Level Imperative Compiler

We use Spatial [29]–a domain-specific language for reconfigurable accelerators–as the front-end of Plasticine. Spatial describes applications with imperative control constructs, such as loops and branches, augmented with parallel patterns [30]. Parallel patterns are transformation functions on memory collections that capture both access patterns and parallelization scheme of the operator. Examples of popular parallel patterns include *map* and *reduce*. Instead of using software data-structures, Spatial exposes hardware memories available on reconfigurable hardware, such as registers and SRAM, directly to programmers. These memory types allow a user to explicitly control data transfer between different levels of memory hierarchies to maximize locality. Additionally, Spatial's language constructs include important design parameters that are essential for achieving good performance on a spatial architecture. Parameters include blocking size, loop unrolling factors, and pipelining schemes, making it easy to perform application-level design space exploration. To enable loop-level parallelization and pipelining, Spatial automatically partitions and buffers the intermediate on-chip memories. An example of outer product—element-wise multiplication of two vectors resulting in a matrix—in Spatial is shown in Figure 2.6.

Spatial is a target-agnostic language for general reconfigurable architectures. The primary targets of Spatial are FPGAs. Similar to the C-based high-level synthesis languages, such as Vivado HLS [31] and SDAccel [32], Spatial provides a high-level programming interface that focuses on the algorithmic implementations of the application, hiding the low-level hardware interfaces and RTL programming from the users. To target Plasticine, we take applications described in Spatial, disabling FPGA-specific transformations, and perform architecture-specific lowering to SARA's IR. The key transformations we take from the Spatial compiler is loop unrolling and memory buffering and partitioning (explained later in **??**). Optimizations, such as retiming and scheduling, are disabled for Plasticine, as they cannot be directly applied. Figure 2.7 summarizes the compiler flow to target FPGAs and Plasticine from Spatial.

Although SARA takes Spatial as front-end, the compilation techniques in SARA can be equally applied to other imperative languages with similar control constructs, such as C-based high-level synthesis language, the backend of Halide IR, the TACO compiler, etc. Using Spatial as our front-end language, however, has a few advantages. Rather than adapting existing languages for processor architectures like in most HLS tools, the design of spatial is engineered specifically for reconfigurable architectures. Spatial's language constructs capture the scheduling scheme that can be

```scala
// 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
Accel {
  // Parallelized by op1
  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
    tileA load vecA(i::i+tsA par ip)
    Foreach(min=0, step=tsB, max=N, par=op2) { j =>
      val tileB = SRAM[T](tsB)
      tileB load vecB(j::j+tsB par ip)
      // 2-D scratchpad
      val tileC = SRAM[T](tsA, tsB)
      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
      matC(i::i+tsA, j::j+tsB par ip) store tileC
    }
  }
}
```

Figure 2.6: Example of Outer Product in Spatial.

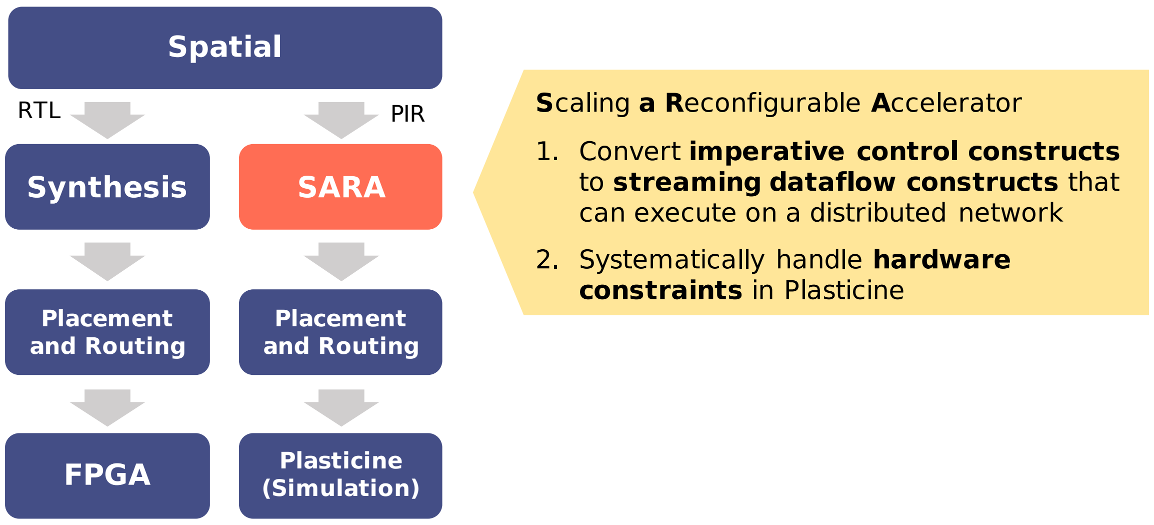

Spatial
RTL
PIR
Synthesis
SARA
Placement and Routing
Placement and Routing
FPGA
Plasticine (Simulation)
Scaling a Reconfigurable Accelerator
1. Convert imperative control constructs to streaming dataflow constructs that can execute on a distributed network
2. Systematically handle hardware constraints in Plasticine

Figure 2.7: Spatial compiler stack to target FPGAs and Plasticine

explored by most spatial architectures, such as coarse-grained pipelining, streaming dataflow, hierarchical parallelization, and finite state machines (FSM). These language constructs are missing from a processor-based language as they cannot be supported. Spatial IR also differs in its representation of control flow and memory constructs, which are more suitable for analyses of spatial architectures.

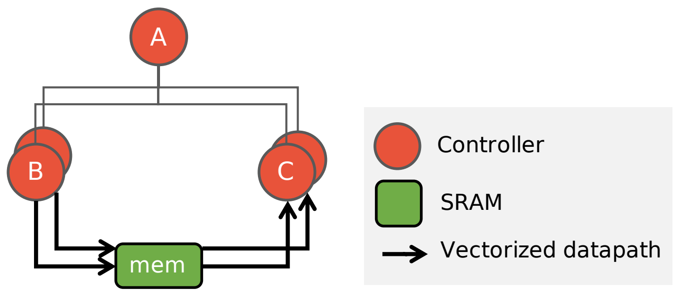
Most compilers, such as LLVM, use control flow graphs to represent the control in an imperative program. The control flow graph implicitly assumes the program is executed in time, which makes it unsuitable for analysis of a reconfigurable architecture that executes the program in space. Spatial uses a control hierarchy in the IR to capture the scheduling of the program. The control hierarchy uses a tree structure to represent the nested control constructs, making it easier to analyze the relations between controllers. Figure 2.8 shows an example of a control flow graph vs. a control hierarchy. The controller at each level of the hierarchy corresponds to a control construct, such as a loop, or a branch statement. A basic block is attached to each *innermost* controller including instructions and memory accesses. If the program has instructions in an outer loop, Spatial automatically inserts a *unit* controller to wrap the floating instructions. SARA takes the backend of Spatial IR as the input, which is a control hierarchy after loop unrolling.

The representation of data structures in traditional IRs often marries to the memory model of a CPU. Traditional compilers treat data collections as pointers to a shared global address space, because CPUs provide a memory abstraction that any data within the global address space can be equally accessible. The hardware, on the back-end, implicitly manage the data movement between an on-chip cache and the off-chip memory; this scheme improves programmability of CPU at the cost of hardware complexity and unpredictability memory performance. Accelerators on the other hand often have explicitly managed on-chip scratchpads. Therefore, modeling the data structures as disjoint memory space is more suitable than pointers for reconfigurable architectures.

Spatial is an embedded DSL in Scala. For simplicity and generality, we will use python-style pseudo code to represents the front-end programming abstraction for the rest of our discussion.

```
mem = array(dims=[100])
A: for i in range(A,par=2):
    B: for j in range(B,vec=16):
        waddr, wdata = ...
        mem(waddr) = wdata # W1
    C: for k in range(C,vec=16):
        raddr = ...
        rdata = mem(raddr) # R1
        ...
```

(a) Pseudo Spatial example



(b) Schematic Spatial IR

Figure 2.8: 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-embedded 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.

Chapter 3

Compiler

3.1 Evaluation (WIP)

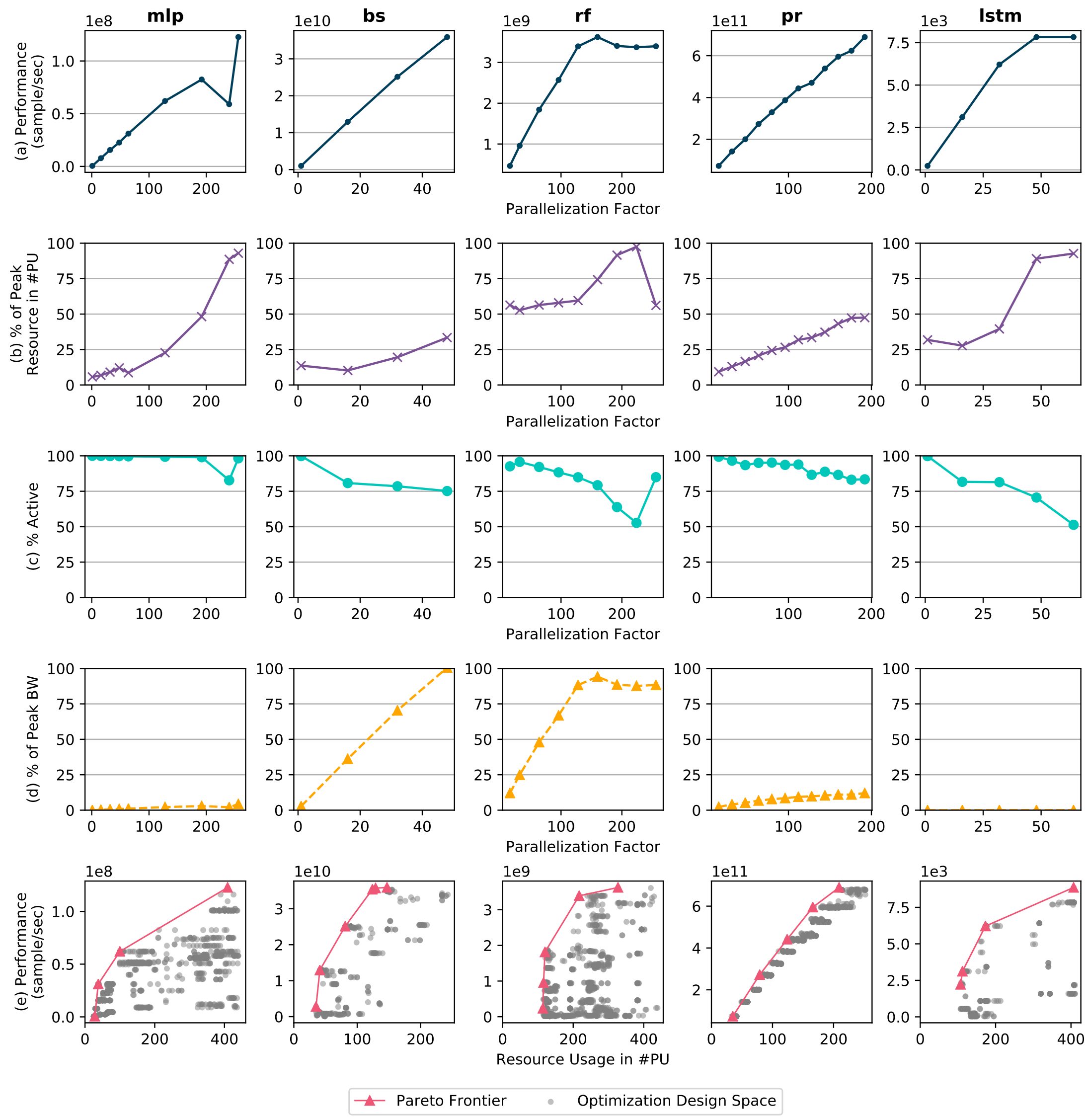


Figure 3.1: Scalability Evaluation. The first four charts show the scaling of (a) throughput, (b) resource usage, (c) runtime activation rate of PUs on the critical path of the compute pipeline, and (d) achieved HBM bandwidth, as the program gets parallelized. (e) shows the combined design space of compiler optimizations and parallelization factors on a throughput-resource curve. The pareto frontier presents the throughput shown in (a) as a function of resource increase in (b).

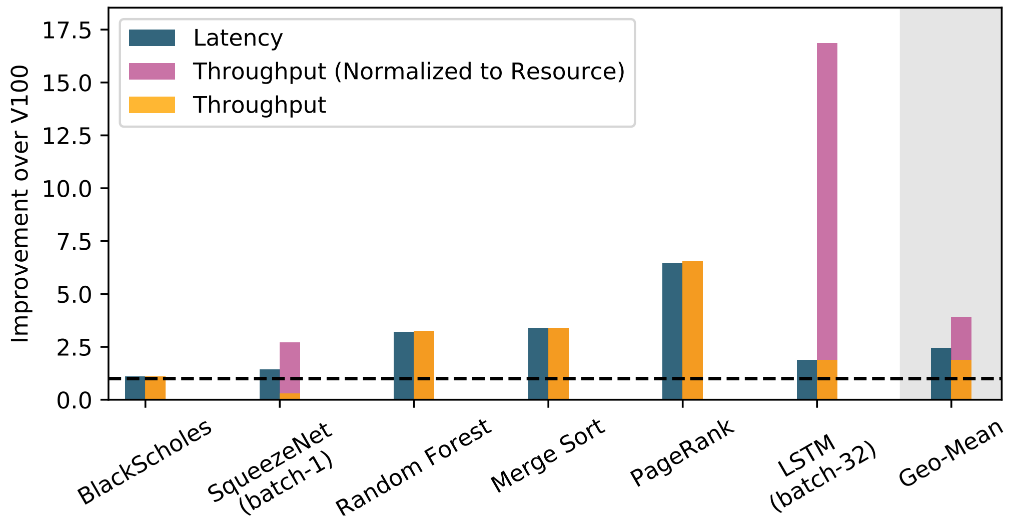


Figure 3.2: Plasticine's latency and throughout improvement over V100 GPU. The evaluated Plasticine architecture has area footprint of $352mm^2$ at 28nm. V100 GPU has area footprint of $815mm^2$ at 12nm. Both platforms have the same off-chip bandwidth at 1TB/s with HBM technology. Yellow and blue bars 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.

Chapter 4

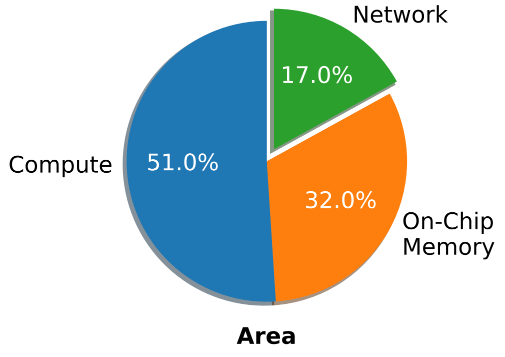
Architecture

In this section, we discuss the architectural advancement on top of the original Plasticine architecture introduced in [17]. 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 4.3 lay outs the datapath changes in order to support more flexible banking schemes required by general access patterns supported in Spatial; Section 4.2 discusses the hardware specialization and architectural sizing for machine learning applications; **??** provides an extensive study on on-chip network selection reconfigurable spatial architectures.

4.1 On-chip Network (Ready)

Achieving scalable performance using spatial architectures while supporting diverse applications requires a flexible, high-bandwidth interconnect. Because modern CGRAs support vector units with wide datapaths, designing an interconnect that balances dynamism, communication granularity, and programmability is a challenging task.

On-chip interconnects can be classified into two broad categories: *static* and *dynamic.* Static interconnects use switches programmed at compile time to reserve high-bandwidth links between communicating units for the lifetime of the application. CGRAs traditionally employ static interconnects [33, 34]. In contrast, dynamic interconnects, or NoCs, contain routers that allow links to be shared between more than one pair of communicating units. NoC communication is typically packet-switched, and routers use allocators to fairly share links between multiple competing



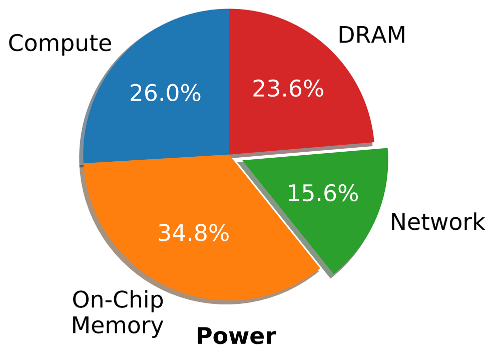


Figure 4.1: Area and power breakdown of Plasticine

packets. Although static networks are fast, they require over-provisioning bandwidth and can be underutilized when a dedicated static link is reserved for a logical link that is not 100% active. While dynamic networks allow link sharing, the area and energy cost to transmit one bit of data is higher for routers than for switches, making bandwidth scaling more expensive in dynamic networks than in static networks.

In this section, we explore the space of spatial architecture interconnect dynamism, granularity, and programmability. We start by characterizing several benchmarks' communication patterns and showing links' imbalanced bandwidth requirements, fanout, and data width in Section 4.1.1. Using these insights, we describe a hybrid network with both static and dynamic capabilities to enable both high bandwidth traffic and high resource sharing. Section 4.1.2 identifies a space of interconnection networks with static and dynamic capabilities, at multiple granularities. Next, We explain our methodology on performance, area, and power modeling in Section 4.1.3. Finally, Section 4.1.4 perform a detailed evaluation across the identified design space for a variety of benchmarks.

Because CGRAs encompass a broad range of architectures, we narrow our study on tiled-based CGRAs with a streaming dataflow execution model, like Plasticine. At high-level, the architecture may contain a pool of heterogeneous compute and memory tiles (we refer as physical units (PUs)) with a global network. The network guarantees exactly-once, in-order delivery with variable latency, and communication between PUs can have varying granularities (e.g., 512-bit vector or 32-bit scalar).

To generalize the study, we introduce a variant processing engine (PE) style than Plasticine's pipeline SIMD unit. The alternative style uses time-scheduled execution, where each PU contains a vector function unit (FU) that can executes a small loop of instructions repeatedly in time. The scheduling window is small enough that instructions are stored as part of the configuration fabric, without dynamic instruction fetch overhead. Compared to the pipelined architecture, this execution model creates more interleaved pipelining across PUs with communication that is tolerant of lower network throughput, which provides an opportunity to share links.

4.1.1 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 [17, 26], 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 to achieve good performance on spatial reconfigurable accelerators is to explore multiple levels of parallelization and pipelining, within and across PUs. By default, pipeline parallelism across PUs introduces high-bandwidth one-to-one communication between dependent stages. To balance the pipeline throughput, we often need to parallize the pipeline stage with the most computation, resulting in one-to-many communication when the receiver stage is parallelized, and many-to-one communication when the producer stage is parallelized. When both the producer and the consumer are parallelized, the worst case is many-to-many communication, as illustrated in **??**. These broadcast and incast patterns introduce challenges in high-performance on-chip network, as their bandwidth demands scale quadratically with chip size in the worst case.

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 [17, 27, 35]. 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

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 [37]	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 4.1: Benchmark summary

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 [36]. 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 PU resources. As the compiler performs optimization passes, such as loop unrolling and memory partitioning, it has static knowledge about communication generated by these transformations. This knowledge enables compiler optimizations to improve network congestion, such as explained in **??**.

To study the communication patterns, 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 4.1 lists the applications and their data size. Figure 4.2 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 [37]), GDA, Kmeans, SGD, and LogReg are compute-intensive applications; for these,

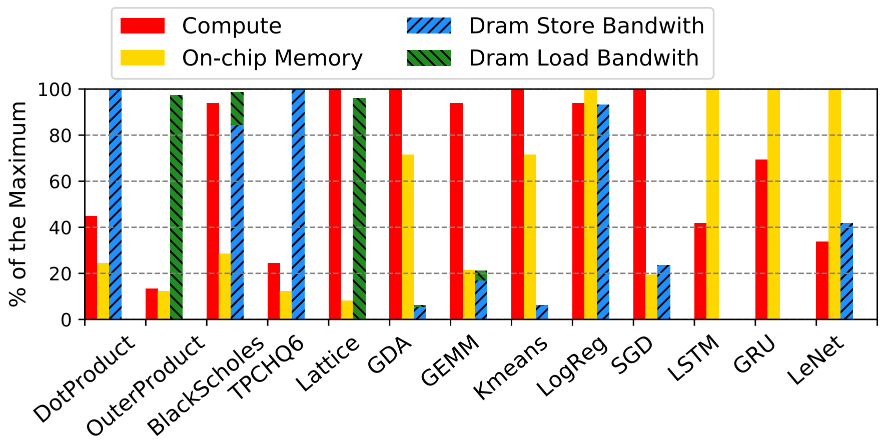


Figure 4.2: Physical resource and bandwidth utilization for various applications.

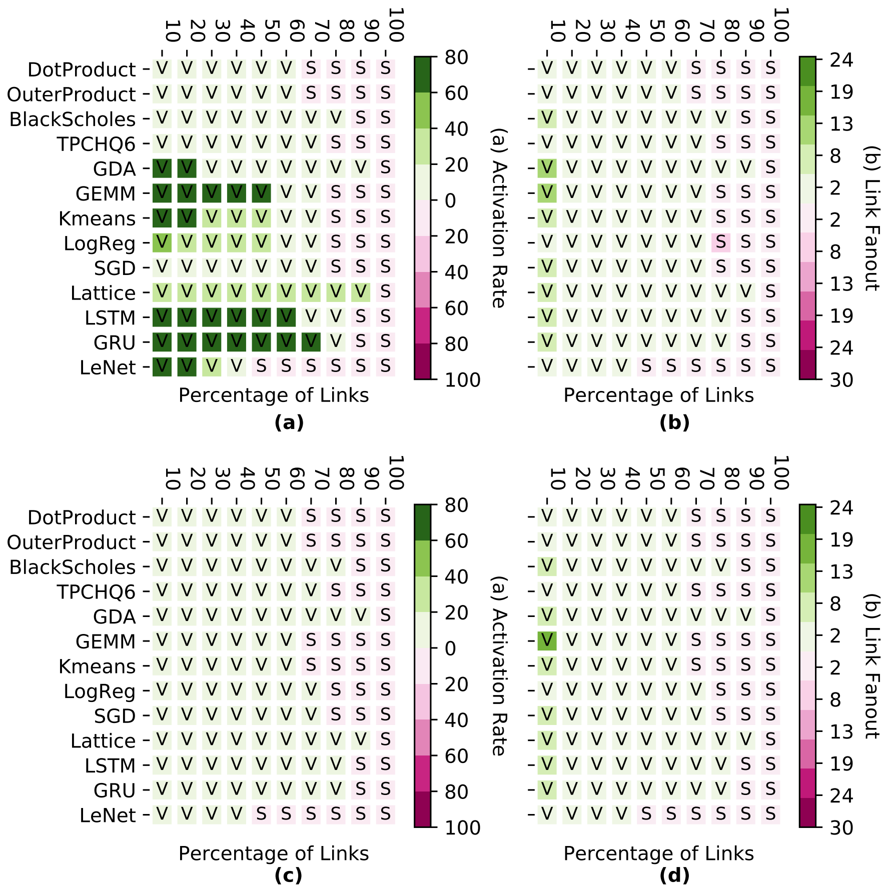


Figure 4.3: Application communication patterns on pipelined (a,b) and scheduled (c,d) CGRA architectures. (a) and (c) show the activation rate distribution of logical links at runtime. Links sorted by granularity, then rate; darker boxes indicate higher rates. The split between green and pink shows the ratio of logical vector to scalar links. (b) and (d) show the distribution of broadcast link fanouts.

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 4.3(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 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 4.3(c,d), exhibit lower bandwidth requirements due to the lower throughput of individual compute PUs. 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 4.4 shows statistics describing the VU dataflow graph (VGDFG) before and after the partitioning described in **??**. The blue bars show the number of VUs, number of logical links, and maximum VU input/output degrees in the original parallelized program; the yellow and green bars show the same statistics after partitioning. Fewer VUs are partitioned for hybrid networks and dynamic networks with the time-scheduled architecture. When a VU in a pipelined CGRA consumes too many inputs or produces too many *distinct* outputs, SARA partitions it to reduce its degree and meet the input/output bandwidth constraints of a purely static network. For dynamic and hybrid networks, partitioning is not strictly necessary, but it improves performance by decreasing congestion at the network ejection port associated with a PU. We do not partition broadcasts with high output degrees because they are handled natively within the network. Finally, the output degree does not change with partitioning because most outputs with a large degree are from broadcast links.

4.1.2 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

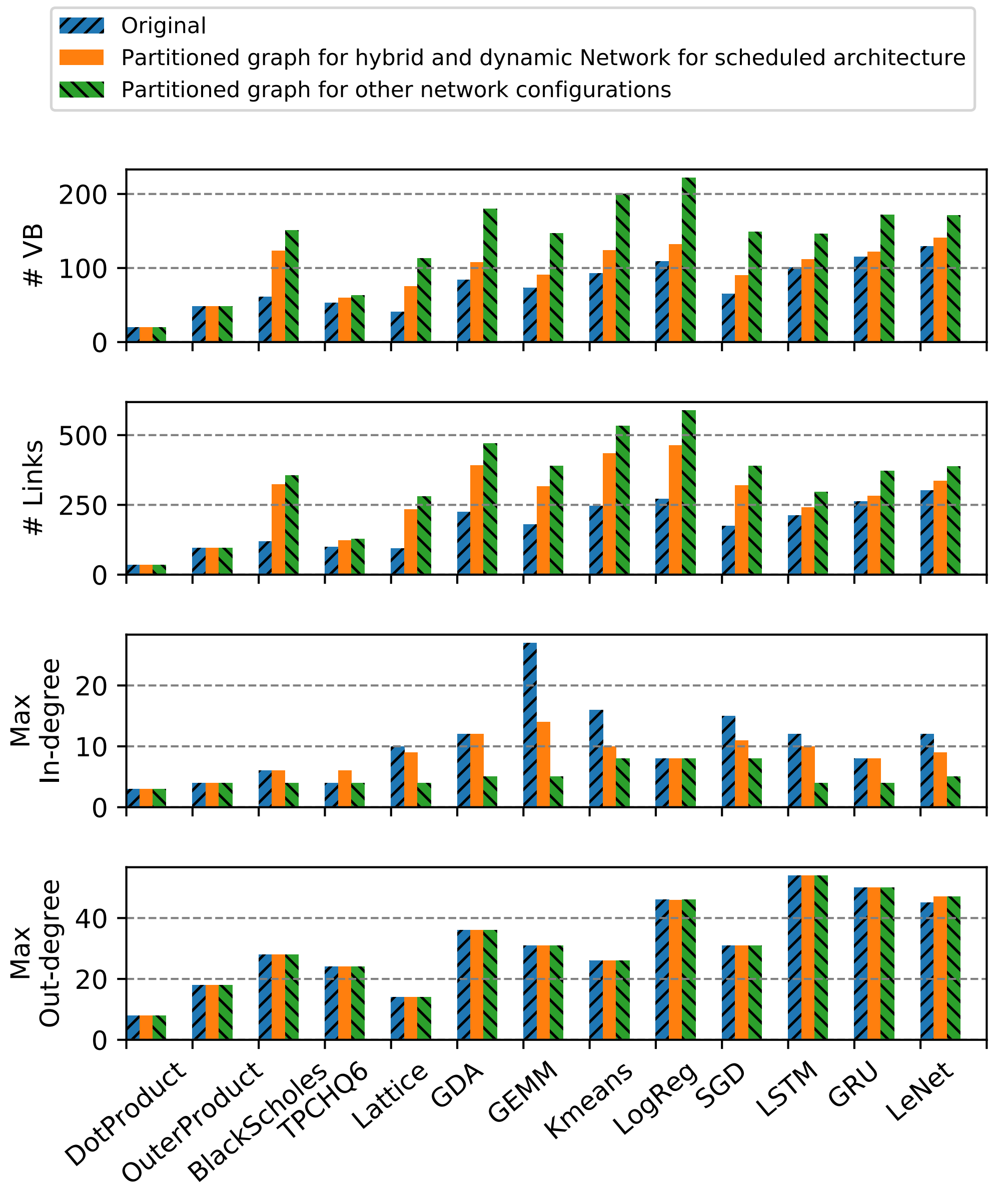


Figure 4.4: Characteristics of program graphs.

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 [38], the source and destination PUs 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 [39]. 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 PU pushes the result to the next PU(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

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 4.2: Network design parameter summary.

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. PUs propagate backpressure signals from their outputs to their inputs, so they must be considered as part of the network graph for deadlock purposes [40]. Furthermore, each PU 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 [38]. 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.

4.1.3 Performance, Area, and Energy Modeling

Next section gives a quantitative analysis of the performance, area, and energy trade-offs involved in choosing a CGRA network, using benchmarks showin in Table 4.1. This section outlines our methodology on how we capture the network performance, area, and energy in our evaluations. Table 4.2 gives our notation for various network parameters.

Simulation

We use a cycle-accurate simulator to model the pipeline and scheduling delay for the two types of architectures, integrated with DRAMSim [41] 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 [42], adding support for arbitrary source routing using look-up tables. Finally, to support efficient multi-casting 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 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 [43] 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 and static power), and active power. The active power, as shown in Section 4.1.3, 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:

$$E_{\text{flit}} = \frac{(P - P_{\text{inactive}})\,T_{\text{testbench}}}{\#\text{flit}} \tag{4.1}$$

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

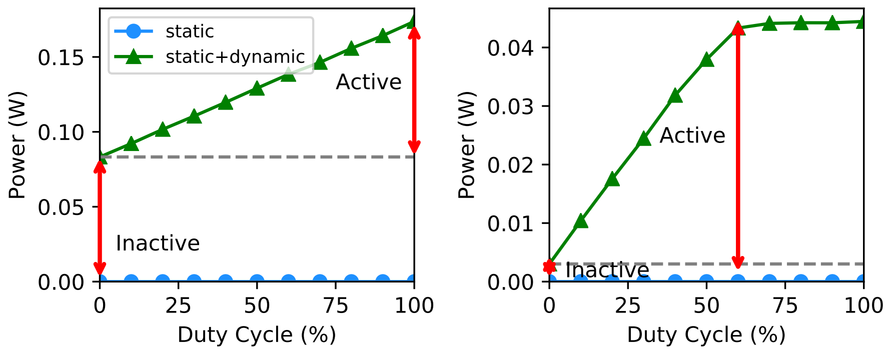


Figure 4.5: Switch and router power with varying duty cycle.

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:

$$E_{\text{net}} = \sum_{\text{allocated}} P_{\text{inactive}} T_{\text{sim}} + E_{\text{flit}} \#\text{flit}, \tag{4.2}$$

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

Figure 4.5 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 [39], 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 4.6. 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 4.5); 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.

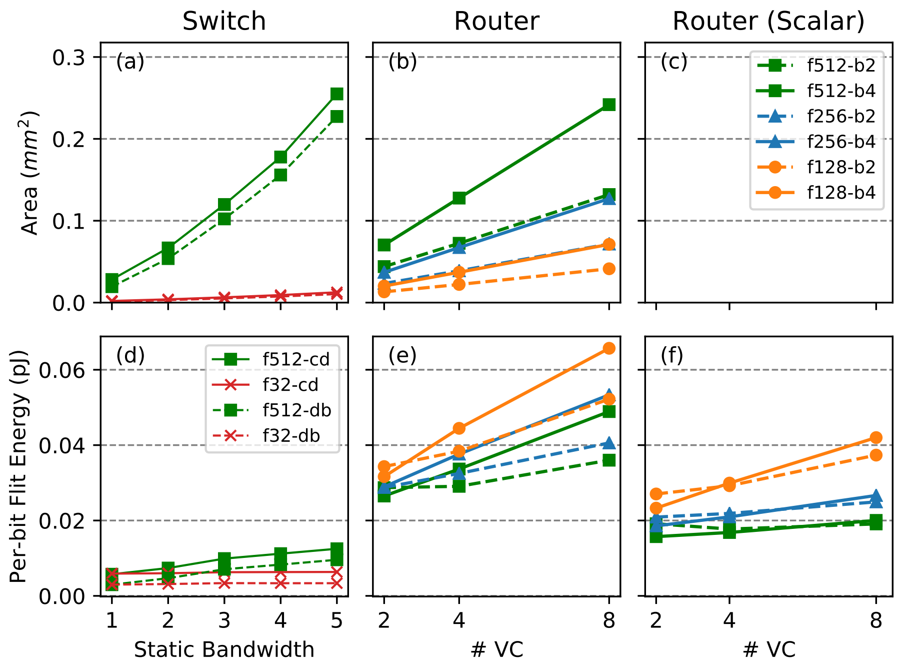


Figure 4.6: Area and per-bit energy for (a,d) switches and (b,c,f) routers. The router only has a vector granularity and can be partially clock-gated when sending scalar packets. Subplots (f) show the energy of the vector router when used for scalar values (32-bit).

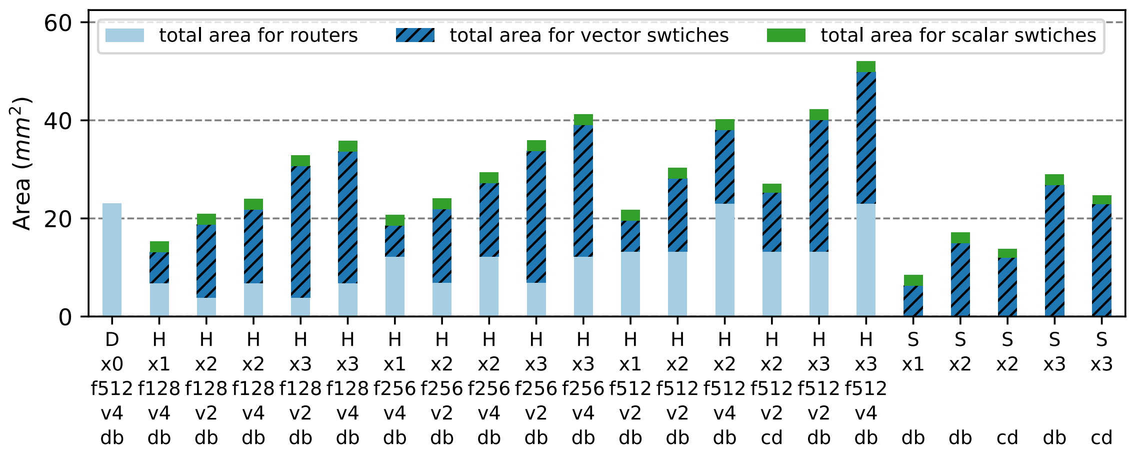


Figure 4.7: Area breakdown for all network configurations.

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 4.7 summarizes the area breakdown of all the network configurations that we evaluate.

4.1.4 Network Architecture Evaluation

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

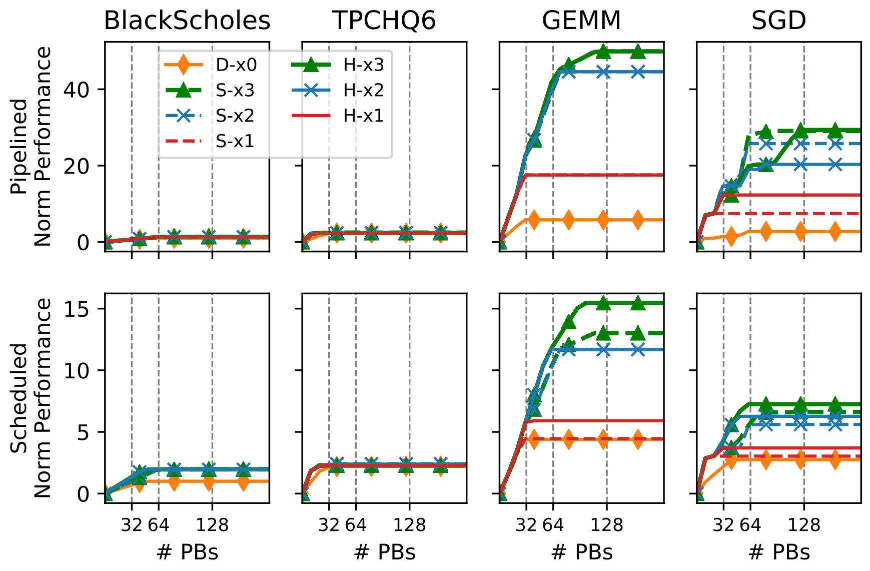


Figure 4.8: Performance scaling with increased CGRA grid size for different networks.

that application. For each network configuration, we present a geometric mean normalized across all applications. For all of our experiments, except Section 4.1.4, we use a network size of 14×14 end-point PUs. All vector networks use a vectorization factor of 16 ($512\,\mathrm{bit}$ messages).

Bandwidth scaling with network size

Figure 4.8 shows how different networks allow several applications to scale to different numbers of PUs. 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 PUs with a small network—which would be underutilized for non-vectorized problems—and smaller compute PUs with limited performance due to network overhead. For more finely grained compute PUs, 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 that of deeply pipelined architectures (top row). Although communication between PUs in these architectures is less frequent, the scheduled architecture must use additional parallelization to match the

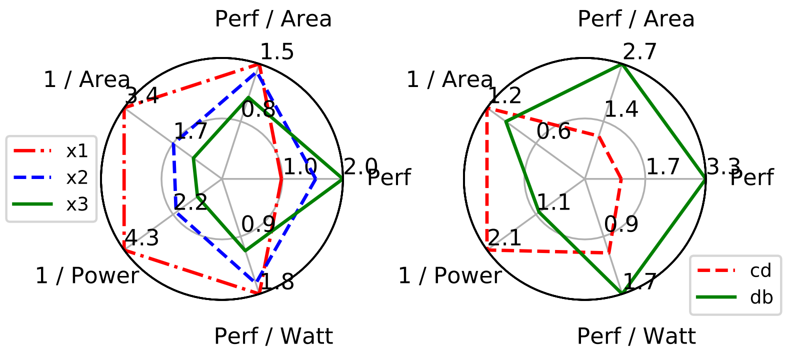


Figure 4.9: Impact of bandwidth and flow control strategies in switches.

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 PU and its router, which prevents the PU 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 4.9, we show that increased static bandwidth results in a linear performance increase and a superlinear increase in area and power. As shown in Section 4.1.4, 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 4.9 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.

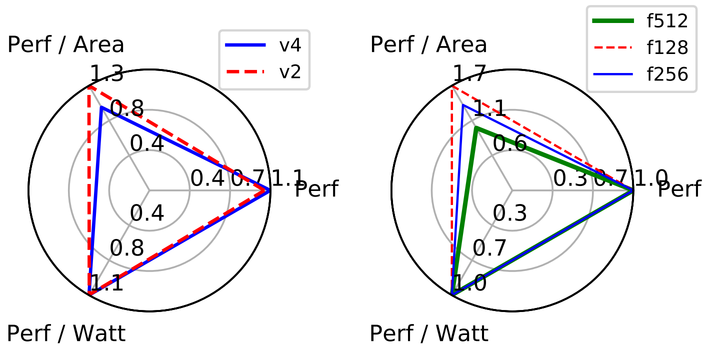


Figure 4.10: Impact of VC count and flit widths in routers.

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 4.1.3, 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 4.10 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 4.11 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 4.1.3, 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 4.10 shows that, for a hybrid network, reducing flit width improves area efficiency with minimal performance loss.

Static vs. hybrid vs. dynamic networks

Figure 4.12 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

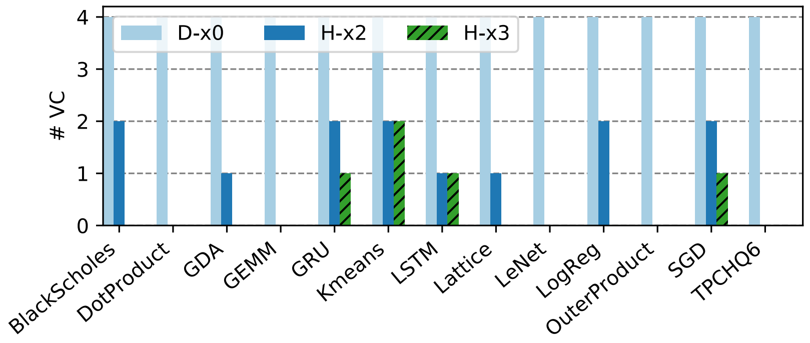


Figure 4.11: Number of VCs required for dynamic and hybrid networks. (No VCs indicates that all traffic is mapped to the static network.)

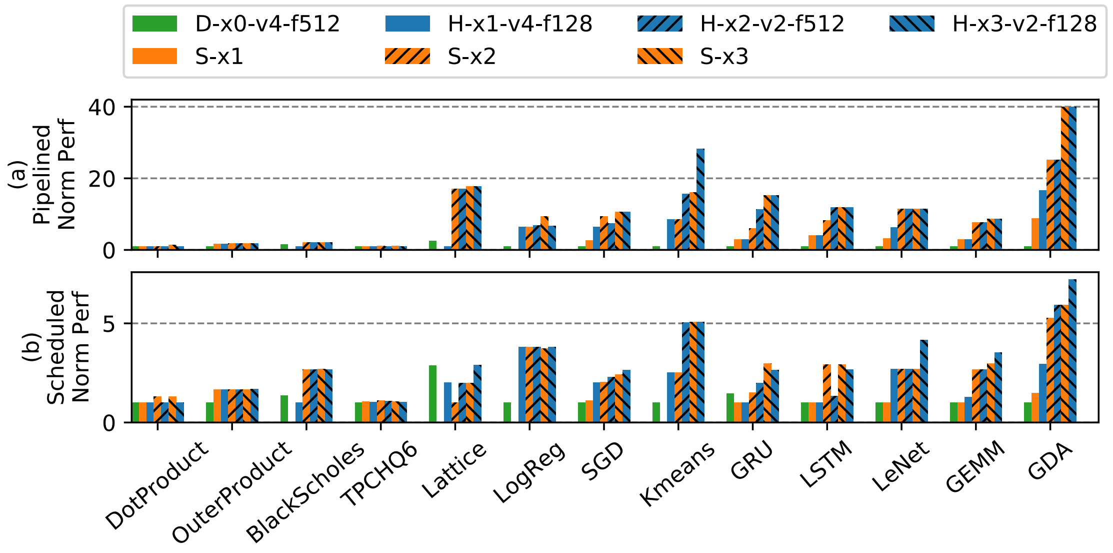


Figure 4.12: Normalized performance for different network configurations.

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 PUs, as shown in Figures 4.13(b,e), because it permits more parallelization. It also has more data movement, as shown in (c,f), because PUs 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 4.13(b,e), where networks D-x0-v4-f512 and S-x2 have small PU 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 4.13(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 PUs are utilized with static networks than hybrid networks. This is because the compiler imposes less stringent IO constraints on PUs when partitioning for the hybrid network (as explained in Section **??**), which results in fewer PUs, less data movement, and greater energy efficiency for hybrid networks.

In Figure 4.14, 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

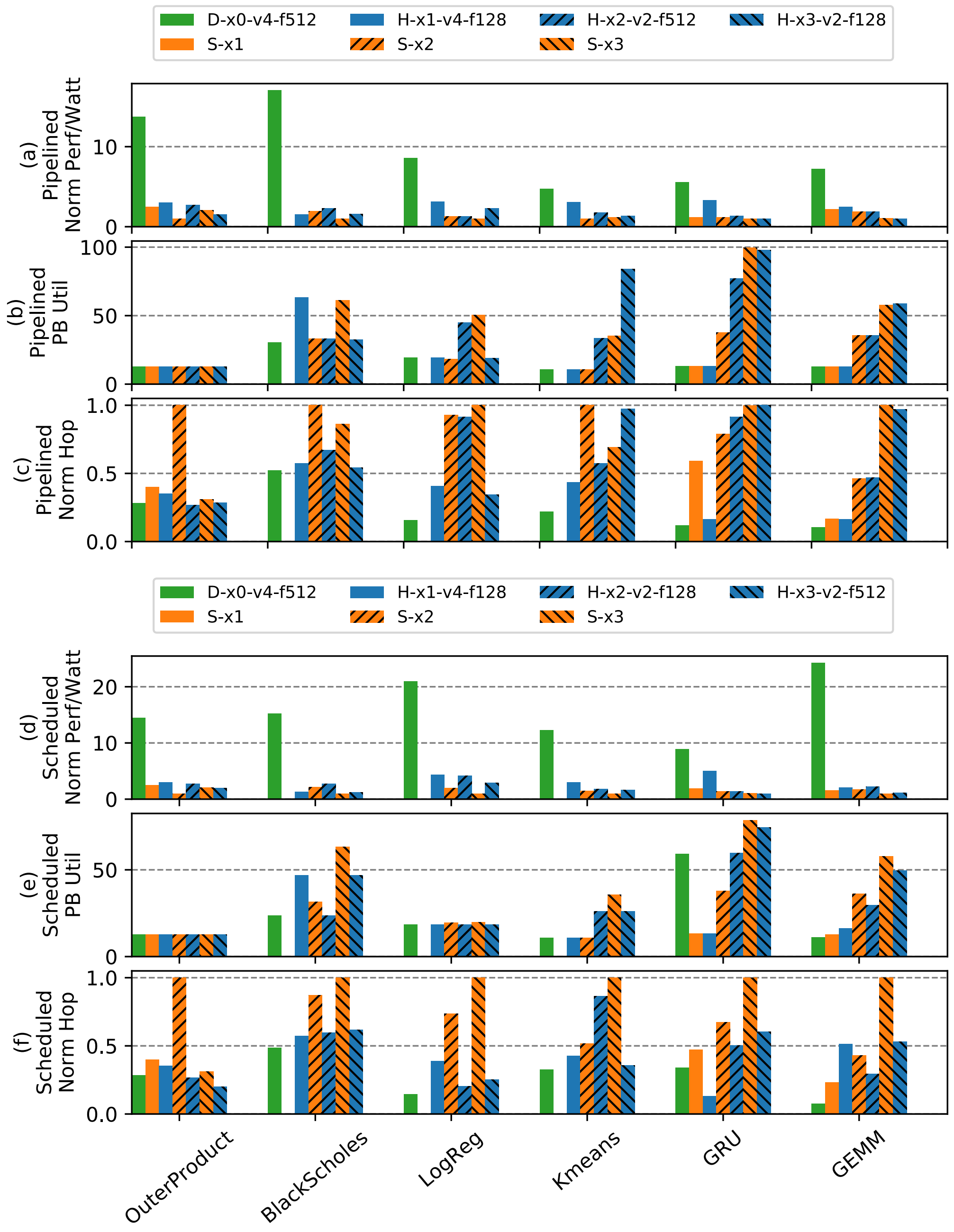


Figure 4.13: (a,d): Normalized performance/watt. (b,e): Percentage of compute and memory PUs utilized for each network configuration. (c,f): Total data movement (hop count).

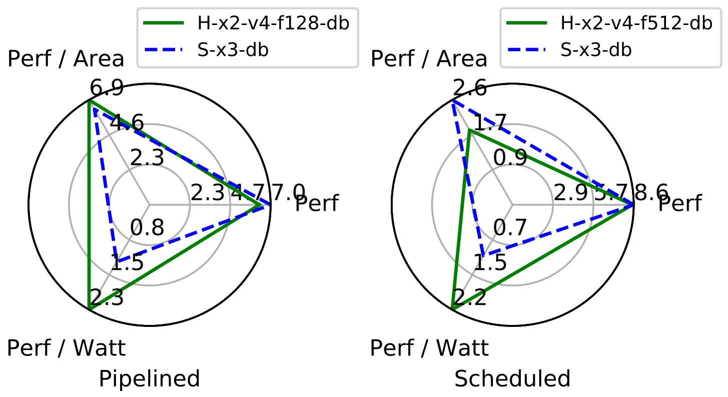


Figure 4.14: Geometric mean improvement for the best network configurations, relative to the worst configuration.

escape path on the dynamic network.

4.2 Plasticine Specialization for RNN Serving (Ready)

Low-precision inference are commonly used to reduce the memory footprint of deep learning models and increase the compute density of hardware accelerators. Plasticine, however, only supports 32-bit operations and datapath. Using RNN serving as a motivating example, this section discusses the necessary architecture augmentation and specialization needed to efficiently map real-time inference on Plasticine.

Recurrent Neural Networks (RNNs) are a class of sequence models that play a key role in low-latency, AI-powered services in datacenters [44, 45]. These applications have stringent tail latency requirements, within the window of milliseconds, for real-time human-computer interactions. An example of such workloads is Google Translate; inference runs concurrently as the user types. Efficient acceleration of RNN requires flexible datapath to support global optimizations beyond BLAS kernels, which is where dataflow accelerators like Plasticine would shine. To meet this low-latency requirement, the other prerequisite is that everything has to stay on-chip. To achieve this, we introduce limited mixed-precision support with changes localized to PCUs. The enhancement supports the commonly used precision in machine learning without introducing massive overhead from fine-grained reconfigurability.

In the rest of this section, Section 4.2.1 proposes the necessary micro-architectural changes to support low-precision arithmetics on Plasticine. Section 4.2.2 introduces a folded reduction structure in

PCU's SIMD that improves the function unit (FU) utilization. Section 4.2.3 discusses architectural parameter selection for Plasticine to serve RNN applications efficiently.

4.2.1 Mixed-Precision Support

Previous works [44, 45] 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 the 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 pack multiple low-precision values into single-precision storage. We support only 8 and 16-bit precisions, which are commonly seen in deep learning inference hardware. Users can only access values that are 32-bit aligned. This constraint guarantees that the microarchitectural change is only local to the PCU. PMU and DRAM access granularity remains intact from the original design.

Figure 4.15 (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, the innermost 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 4.15 enable reduction operations. To support 8-bit element-wise multiplication and 16-bit reduction, we add 4 opcodes to the FU, shown in Figure 4.15 (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 4.15 (a) to complete the remaining reduction and accumulates in a 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 a 32-bit value reduces the two registers in each line into 4 32-bit values. These values are fed through

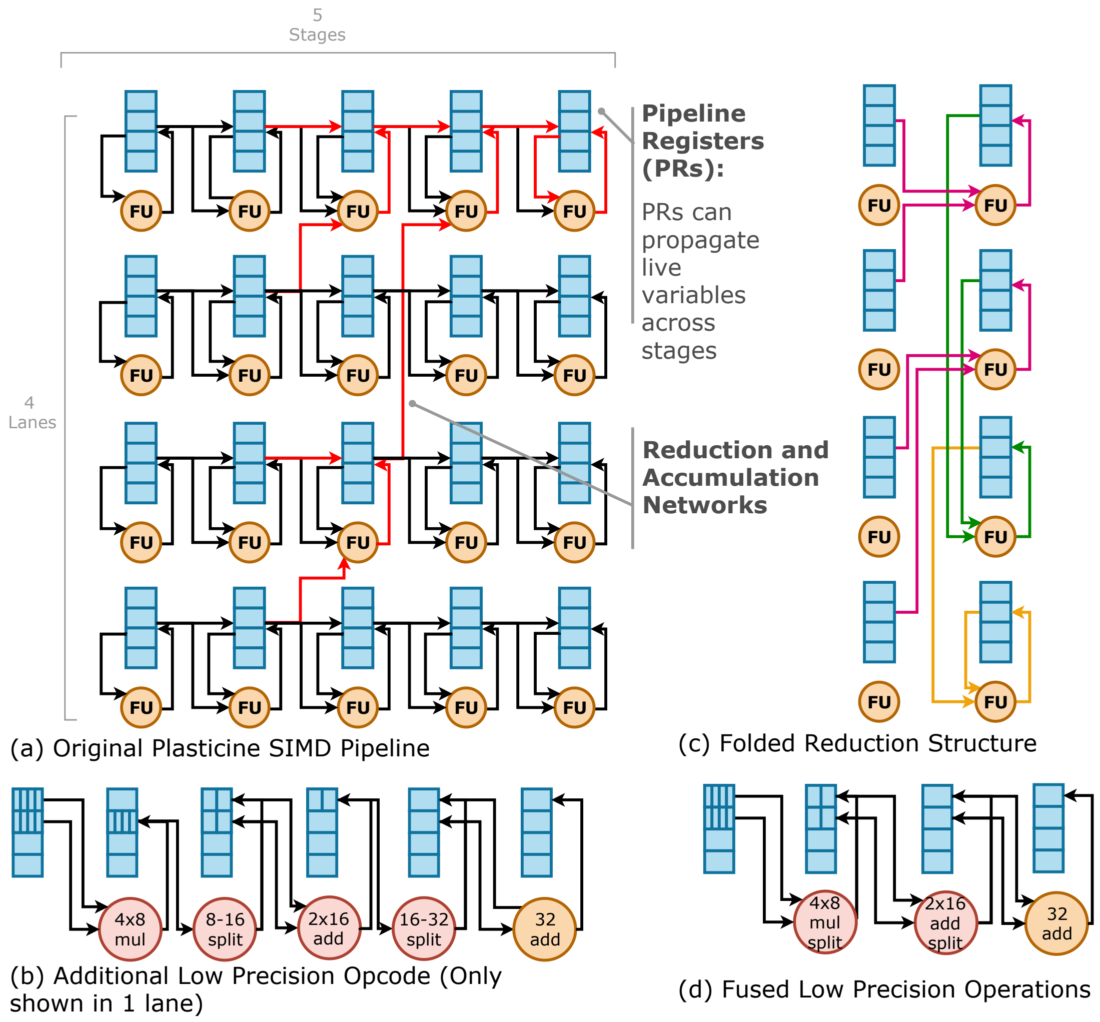

5
Stages
4
Lanes
FU
Pipeline Registers (PRs):
PRs can propagate live variables across stages
Reduction and Accumulation Networks
(a) Original Plasticine SIMD Pipeline
(c) Folded Reduction Structure
4x8 mul
8-16 split
2x16 add
16-32 split
32 add
(b) Additional Low Precision Opcode (Only shown in 1 lane)
4x8 mul split
2x16 add split
(d) Fused Low Precision Operations

Figure 4.15: 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 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 rearrange into the same stage. We also fuse the first low-precision reduction with the next rearrange shown in Figure 4.15 (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.

4.2.2 Folded Reduction Tree

The original reduction tree shown in Figure 4.15 (a) requires $\log_2(\#_{LANE}) + 1$ number of stages for $\#_{LANE}$ operations, leading to a low utilization of the FUs in the SIMD pipeline. The reduction tree also restricts the SIMD pipeline to have at least $\log_2(\#_{LANE}) + 1$ stages to compute a full reduction within a PCU. For a SIMD pipeline with 16 lanes, the FU utilization is only 53.33% in reduction stages.

To improve FU utilization, we introduce a folded reduction structure that performs $\#_{LANE}$ operations entirely within a single-stage pipelined over multiple cycles. Figure 4.15 (c) shows the folded reduce-accumulate structure. Instead of feeding the output of the reduction operation to the next stage, this structure folds the output back to the PR of the next unused FU in the same stage. The entire reduction plus accumulation is still fully pipelined in $\log_2(\#_{LANE}) + 1$ cycles with no structural hazard. In the original design, only a single register among the PRs have the special reduction tree connection. The downside of the folded structure is that no other live variables can be propagated after this reduction stage, unless adding multiple folded trees. In applications, we rarely see the need for multiple live variables after the reduction operation other than the accumulator itself. Therefore, it makes the most sense to put the folded reduction tree only in the last stage of a SIMD pipeline.

With the fused reduced-precision operations and the folded reduction tree, a PCU is able to perform 64 8-bit map-reduce in 4 stages, and 32 16-bit map-reduce in 3 stages. All the operations are still completed in $2 + \log_2(\#_{LANE})$ cycles, i.e. 8, 7, and 6 cycles for 8-, 16-, and 32-bit operations, respectively.

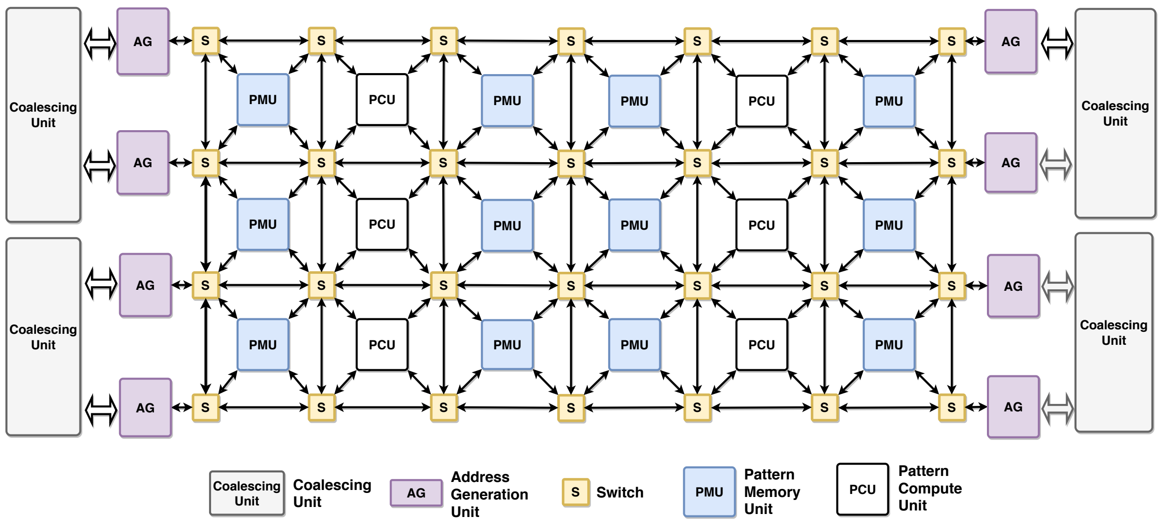


Figure 4.16: Variant Plasticine configuration for RNN serving with 2:1 ratio for PMU and PCU

4.2.3 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 a 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 six stages and 16 lanes, and a PMU has 16 banks, which gives 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. Figure 4.16 shows a specialized Plasticine configuration for RNN serving and general machine learning. Specifically, we choose a 2 to 1 PMU-PCU ratio with 4 stages in each PCU.

4.3 Generic Banknig Support (Ready)

To support Spatial's generic banking scheme for on-chip SRAM mentioned in **??**, we need to introduce a few microarchitectural changes to Plasticine's datapath. As a background, the static banking analysis searches an address remapping scheme, which we refer as banking scheme, that remaps the logical address space into partitions, such that addresses accessed in parallel belongs to different partitions. Each partition corresponds to a physical memory bank that comes with an additional address port. If the compiler can find a way to partition the data such that all parallel workers are sent to different physical banks, the memory is guaranteed to feed all workers at full-throughput, which scales performance with parallelism. To search for the banking schemes, the compiler analyzes the access patterns of the memory. An access pattern refers to groups of address expressions that can be queried concurrently. For each access pattern, there can be multiple banking schemes that can sustain the access bandwidth. However, not all schemes cost the same amount of resources; some are more expensive in logic, and others are more expensive in memory.

The partitioning logic are two parametrizable equations for the bank address (BA) and the bank offset (BO), both are functions of the original user-requested logical addresses. BA is an expression that selects the partition for each request, and BO is the offset within the partition. The banking analyzer searches the parameters in the equation and injects these computations to memory accesses in the program. The first addition to Plasticine is the ability to compute independent BI and BO for each vector lanes. We introduce vectorized address pipelines in PMUs, which was originally a scalar pipeline.

The generic banking scheme also requires a full crossbar datapath between all access lanes (workers) and physical banks. To support this, we introduce the shuffle operator shown in **??**. The shuffle operator takes three vectorized operands: a from address $\overrightarrow{FA}$, a to address $\overrightarrow{TA}$, and a base $\overrightarrow{B}$. The output O equals to B shuffled from the order specified in $\overrightarrow{FA}$ to the order in $\overrightarrow{BA}$. Specifically,

$$\forall i \in [1, |\overrightarrow{O}|], O_i = \begin{cases} B_j, \text{if } \exists\, TA_i = FA_j \\ 0 \end{cases} \tag{4.3}$$

$$|\overrightarrow{FA}| = |\overrightarrow{B}| \tag{4.4}$$

$$|\overrightarrow{TA}| = |\overrightarrow{O}| \tag{4.5}$$

?? gives concrete examples of inputs and outputs of the shuffle operator. On the requester side, BA is the FA, a vector constant corresponding to the banks statically assigned to the partition is the TA, and BO is the base. On the receiver side, the vector constant is the FA, the BA is the TA, and the data response D is the base. If a lane in the TA is not in the FA, the corresponding lane is marked as invalid with value 0. As a result, an invalid vector ORed with a valid vector is still the valid vector. For address going to the scratchpad, we use the first bit of the 32-bit address as the predication bit of the access; if the predicate bit equals zero, the request will be dropped by the memory. Therefore, a valid address zero would be xF0000000 to distinguish with an invalid address 0.

The shuffle operator can be expensive, requiring a 16×16 32-bit crossbar with parallel integer comparators for equality. We expect to add one or two shuffle operators per SIMD pipeline, and the operator can be pipelined across multiple stages to meet timing.

Chapter 5

Related Work

5.1 Network (Ready)

Multiple decades of research have resulted in a rich body of literature, both in CGRAs [33, 34] and on-chip networks [46]. We discuss relevant prior work under the following categories:

5.1.1 Tiled Processor Interconnects

Architectures such as Raw [47] and Tile [48] use scalar operand networks [49], 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 [50] is a tiled dataflow architecture with dynamic execution. TRIPS does not have a static interconnect, but contains two dynamic networks [51]: an operand network to route operands between tiles, and an on-chip network to communicate with cache banks. Wavescalar [52] 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 [53] 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 [54]. CGRAs such as HRL [24], DySER [21], and Elastic CGRAs [55] 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 [56, 57, 58] 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 [25] 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 [59] 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 [60] focuses on area and energy tradeoffs with different network topologies with the ADRES [19] architecture, where all topologies use a fully static interconnect. KressArray Xplorer [61] similarly explores topology tradeoffs with the KressArray [20] architecture. Other studies explore topologies for mesh-based CGRAs [62] and more general CGRAs supporting resource sharing [63]. Other tools like Sunmap [64] allow end users to construct and explore various topologies.

5.1.4 Compiler Driven NoCs (WIP)

Other prior works have used compiler techniques to optimize various facets of NoCs. Some studies have explored statically allocating virtual channels [65, 66] 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 [67]. 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. [68] 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. [69] use a static placement algorithm and an estimate of reliability to attempt to guide placement decisions for NoCs. Kapre et al. [70] 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 [71] 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 [72, 73, 74] are not able to extract nested loop-level parallelism from modern data-intensive applications. For example, StreamIt [72], 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 [21] and Tartan [23]) 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) [75] can sustain higher interconnect bandwidth by introducing on-chip hierarchy; however, it lacks support for polyhedral memory banking [76], a pivotal optimization to achieve massive parallel accesses to on-chip memory. Plasticine [17] 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 [77, 78] 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 [17] 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 **??** and **??**, the Plasticine compiler allocates one VB for a specific type of PB and under-utilizes resources within PBs.

Chapter 6

Conclusions (WIP)

We show that the best network design depends on both applications and the underlying accelerator architecture. Network performance correlates strongly with bandwidth for streaming accelerators, and scaling raw bandwidth is more area- and energy-efficient with a static network. We show that the application mapping can be optimized to move less data by using a dynamic network as a fall-back from a high-bandwidth static network. This static-dynamic hybrid network provides a 1.8x energy-efficiency and 2.8x performance advantage over the purely static and purely dynamic networks, respectively.

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

```
bufferA = VecSteram()
bufferB = ScalarStream()
bufferC = VecSteram()
outputD = VecSteram()

with Context() as ctx:
    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(expr)
```

(b) Imperative configuration

Bibliography

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted mosfet’s with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, Oct 1974.

[2] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, “Understanding sources of inefficiency in general-purpose chips,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA ’10, (New York, NY, USA), pp. 37–47, ACM, 2010.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pp. 365–376, June 2011.

[4] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture,” *Commun. ACM*, vol. 62, pp. 48–60, Jan. 2019.

[5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” *SIGARCH Comput. Archit. News*, vol. 45, pp. 1–12, June 2017.

[6] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 243–254, IEEE Press, 2016.

[7] T. Zhao, Y. Zhang, and K. Olukotun, "Serving recurrent neural networks efficiently with a spatial accelerator," in *Proceedings of the 2 nd SysML Conference*, 2019.

[8] I. Kuon, R. Tessier, and J. Rose, "Fpga architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, pp. 135–253, 01 2007.

[9] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 13–24, IEEE Press, 2014.

[10] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, and S. Jiang, "Sda: Software-defined accelerator for largescale dnn systems," Hot Chips 26, 2014.

[11] K. Guo, L. Sui, J. Qiu, S. Yao, S. Han, Y. Wang, and H. Yang, "From model to fpga: Software-hardware co-design for efficient neural network acceleration," in *2016 IEEE Hot Chips 28 Symposium (HCS)*, pp. 1–27, Aug 2016.

[12] A. AWS, "Amazon ec2 f1 instances." `https://aws.amazon.com/ec2/instance-types/f1`.

[13] I. Bolsens, "Programming modern fpgas, international forum on embedded multiprocessor soc, keynote,." `http://www.xilinx.com/univ/mpsoc2006keynote.pdf`, 2006.

[14] B. H. Calhoun, J. F. Ryan, S. Khanna, M. Putic, and J. Lach, "Flexible circuits and architectures for ultralow power," *Proceedings of the IEEE*, vol. 98, pp. 267–282, Feb 2010.

[15] K. K. W. Poon, S. J. E. Wilton, and A. Yan, "A detailed power model for field-programmable gate arrays," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, pp. 279–302, Apr. 2005.

[16] I. Kuon, R. Tessier, and J. Rose, "Fpga architecture: Survey and challenges," *Found. Trends Electron. Des. Autom.*, vol. 2, pp. 135–253, Feb. 2008.

[17] 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*, ISCA '17, (New York, NY, USA), pp. 389–402, ACM, 2017.

[18] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered instructions: A control paradigm for spatially-programmed architectures," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 142–153, ACM, 2013.

[19] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *Field Programmable Logic and Application* (P. Y. K. Cheung and G. A. Constantinides, eds.), (Berlin, Heidelberg), pp. 61–70, Springer Berlin Heidelberg, 2003.

[20] R. Kress, "A fast reconfigurable alu for xputers," 1996.

[21] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, (Washington, DC, USA), pp. 503–514, IEEE Computer Society, 2011.

[22] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "Piperench: a coprocessor for streaming multimedia acceleration," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, pp. 28–39, May 1999.

[23] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: Evaluating spatial computation for whole program execution," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 163–174, ACM, 2006.

[24] M. Gao and C. Kozyrakis, "Hrl: Efficient and flexible reconfigurable logic for near-data processing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 126–137, March 2016.

[25] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, (New York, NY, USA), pp. 45:1–45:6, ACM, 2017.

[26] J. Noguera, C. Dick, V. Kathail, G. Singh, K. Vissers, and R. Wittig, "Xilinx project everest: 'hw/sw programmable engine'," Hot Chips 30, 2018.

[27] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale DNN processor for real-time AI," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pp. 1–14, 2018.

[28] J. von Neumann, "First draft of a report on the edvac," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[29] 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.

[30] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, (Washington, DC, USA), pp. 89–100, IEEE Computer Society, 2011.

[31] "Vivado high-level synthesis." `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, 2016.

[32] Xilinx, "The xilinx sdaccel development environment." `https://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf`, 2014.

[33] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proceedings of the IEEE*, vol. 103, pp. 332–354, March 2015.

[34] K. Choi, "Coarse-grained reconfigurable array: Architecture and application mapping," *IPSJ Transactions on System LSI Design Methodology*, vol. 4, pp. 31–46, 2011.

[35] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 416–429, June 2017.

[36] D. Sanchez, G. Michelogiannakis, and C. Kozyrakis, "An analysis of on-chip interconnection networks for large-scale chip multiprocessors," *ACM Trans. Archit. Code Optim.*, vol. 7, pp. 4:1–4:28, May 2010.

[37] E. Garcia and M. Gupta, "Lattice regression," in *Advances in Neural Information Processing Systems*, pp. 594–602, 2009.

[38] X. Wang, P. Liu, M. Yang, and Y. Jiang, "Avoiding request–request type message-dependent deadlocks in networks-on-chips," *Parallel Computing*, vol. 39, no. 9, pp. 408–423, 2013.

[39] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. Elsevier, 2004.

[40] A. Hansson, K. Goossens, and A. Rădulescu, "Avoiding message-dependent deadlock in network-based systems on chip," *VLSI design*, vol. 2007, 2007.

[41] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "Dramsim: A memory system simulator," *SIGARCH Comput. Archit. News*, vol. 33, pp. 100–107, Nov. 2005.

[42] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 86–96, IEEE, 2013.

[43] D. U. Becker, *Efficient Microarchitecture for Network-on-Chip Routers*. PhD thesis, Stanford University, Palo Alto, 2012.

[44] 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.

[45] 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.

[46] N. E. Jerger, T. Krishna, and L.-S. Peh, "On-chip networks, second edition," *Synthesis Lectures on Computer Architecture*, vol. 12, no. 3, pp. 1–210, 2017.

[47] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, pp. 25–35, Mar. 2002.

[48] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, pp. 15–31, Sept. 2007.

[49] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar operand networks: On-chip interconnect for ilp in partitioned architectures," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, (Washington, DC, USA), pp. 341–353, IEEE Computer Society, 2003.

[50] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pp. 422–433, June 2003.

[51] P. Gratz, C. Kim, K. Sankaralingam, H. Hanson, P. Shivakumar, S. W. Keckler, and D. Burger, "On-chip interconnection networks of the trips chip," *IEEE Micro*, vol. 27, pp. 41–50, Sept. 2007.

[52] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The wavescalar architecture," *ACM Trans. Comput. Syst.*, vol. 25, pp. 4:1–4:54, May 2007.

[53] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 370–380, Dec 2009.

[54] A. Ye and J. Rose, "Using bus-based connections to improve field-programmable gate-array density for implementing datapath circuits," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 14, pp. 462–473, May 2006.

[55] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic cgras," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, (New York, NY, USA), pp. 171–180, ACM, 2013.

[56] B. Van Essen, A. Wood, A. Carroll, S. Friedman, R. Panda, B. Ylvisaker, C. Ebeling, and S. Hauck, "Static versus scheduled interconnect in coarse-grained reconfigurable arrays," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 268–275, IEEE, 2009.

[57] G. Dimitroulakos, M. D. Galanis, and C. E. Goutis, "Exploring the design space of an optimized compiler approach for mesh-like coarse-grained reconfigurable architectures," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10–pp, IEEE, 2006.

[58] C. Nicol, "A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing."

[59] C. A. Moritz, D. Yeung, and A. Agarwal, "Exploring optimal cost-performance designs for raw microprocessors," in *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, pp. 12–27, April 1998.

[60] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev, "Architectural exploration of the adres coarse-grained reconfigurable array," in *Reconfigurable Computing: Architectures, Tools and Applications* (P. C. Diniz, E. Marques, K. Bertels, M. M. Fernandes, and J. M. P. Cardoso, eds.), (Berlin, Heidelberg), pp. 1–13, Springer Berlin Heidelberg, 2007.

[61] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Kressarray xplorer: A new cad environment to optimize reconfigurable datapath array architectures," in *Proceedings 2000. Design Automation Conference (DAC)*, pp. 163–168, Jan 2000.

[62] N. Bansal, S. Gupta, N. Dutt, A. Nicolau, and R. Gupta, "Network topology exploration of mesh-based coarse-grain reconfigurable architectures," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, pp. 474–479, Feb 2004.

[63] Y. Kim, R. N. Mahapatra, and K. Choi, "Design space exploration for efficient resource utilization in coarse-grained reconfigurable architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, pp. 1471–1482, Oct 2010.

[64] S. Murali and G. D. Micheli, "Sunmap: a tool for automatic topology selection and generation for nocs," in *Proceedings. 41st Design Automation Conference, 2004.*, pp. 914–919, July 2004.

[65] M. A. Kinsy, M. H. Cho, T. Wen, E. Suh, M. van Dijk, and S. Devadas, "Application-aware deadlock-free oblivious routing," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 208–219, ACM, 2009.

[66] K. S. Shim, M. H. Cho, M. Kinsy, T. Wen, M. Lis, G. E. Suh, and S. Devadas, "Static virtual channel allocation in oblivious routing," in *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pp. 38–43, May 2009.

[67] C. J. Glass and L. M. Ni, "The turn model for adaptive routing," in *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, pp. 278–287, May 1992.

[68] O. Ozturk, M. Kandemir, M. J. Irwin, and S. H. Narayanan, "Compiler directed network-on-chip reliability enhancement for chip multiprocessors," *ACM Sigplan Notices*, vol. 45, no. 4, pp. 85–94, 2010.

[69] C. Ababei, H. S. Kia, O. P. Yadav, and J. Hu, "Energy and reliability oriented mapping for regular networks-on-chip," in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, pp. 121–128, ACM, 2011.

[70] N. Kapre and A. Dehon, "An NoC traffic compiler for efficient FPGA implementation of sparse graph-oriented workloads," *International Journal of Reconfigurable Computing*, vol. 2011, 2011.

[71] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (Berkeley, CA, USA), pp. 265–283, USENIX Association, 2016.

[72] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, (New York, NY, USA), pp. 291–303, ACM, 2002.

[73] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik, “Chlorophyll: Synthesis-aided compiler for low-power spatial architectures,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’14, (New York, NY, USA), pp. 396–407, ACM, 2014.

[74] N. Trifunovic, H. Palikareva, T. Becker, and G. Gaydadjiev, “Cloud deployment and management of dataflow engines,” in *Proceedings of the 1st International Workshop on Next Generation of Cloud Architectures*, CloudNG:17, (New York, NY, USA), pp. 5:1–5:6, ACM, 2017.

[75] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, “Towards general purpose acceleration by exploiting common data-dependence forms,” in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’52, (New York, NY, USA), pp. 924–939, ACM, 2019.

[76] Y. Wang, P. Li, and J. Cong, “Theory and algorithm for generalized memory partitioning in high-level synthesis,” in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA ’14, (New York, NY, USA), pp. 199–208, ACM, 2014.

[77] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, “A general constraint-centric scheduling framework for spatial architectures,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’13, (New York, NY, USA), pp. 495–506, ACM, 2013.

[78] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, “Spatial computation,” in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, (New York, NY, USA), pp. 14–26, ACM, 2004.