DESIGN OF PROGRAMMABLE, ENERGY-EFFICIENT RECONFIGURABLE ACCELERATORS

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

Raghu Prabhakar
August 2018

This dissertation is online at: http://purl.stanford.edu/dj143cx9118

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Chris Re

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Current trends in technology scaling, coupled with the increasing compute demands with a limited power budget, has spurred research into specialized accelerator architectures. Field-Programmable Gate Arrays (FPGAs) have gained traction in the past few years as energy-efficient accelerator subtrates as they avoid the energy overheads of instruction-based processor architectures with a statically programmable data path. FPGA architectures support fine-grained reconfigurability at the bit level, which provides flexibility required to implement arbitrary state machines and data paths. However, bit-level reconfigurability in FPGAs creates *programming* inefficiencies; low-level programming models limit the accessibility of FPGAs to expert hardware designers, and complicates the compiler flow which often takes several hours. Furthermore, bit-level reconfigurability also creates *architectural* inefficiencies, which increases area and power overheads while reducing compute density.

This dissertation addresses both programming and architectural inefficiencies of FPGAs, by describing a compiler flow and a new reconfigurable architecture based on high-level parallel patterns. Parallel patterns are high-level programming abstractions underlying several domain-specific languages (DSLs) that capture parallelism, memory access information and data locality in applications. To address programming inefficiencies, a new representation based on composable, parameterized hardware templates is proposed that is designed to be targeted from parallel patterns. These templates are designed to capture nested parallelism and locality in applications, and parameterized to expose the application design space to the compiler. A compiler flow is described that performs two key transformations: *tiling*, and *metapipelining*, to automatically translate parallel patterns to templates, and hardware. Evaluation of the compiler framework on a Stratix V FPGA

shows speedups of up to 39.4× over an optimized baseline. To address architectural inefficiencies, this dissertation proposes a new coarse-grained reconfigurable architecture (CGRA) called Plasticine. Plasticine is built with reconfigurable primitives that natively exploits SIMD, pipelined parallelism, and coarse-grained parallelism at multiple levels. A configurable on-chip memory system with programmable address generation, address interleaving across banks, and buffering enables efficiently exploiting data locality and sustain compute throughput for various access patterns. Pipelined, static interconnects at multiple bus widths allow communication at multiple granularities while minimizing area overhead. Dedicated off-chip address generators and scatter-gather units maximize DRAM bandwidth utilization for dense and sparse accesses. With an area footprint of $113mm^2$ in a 28-nm process and a 1-GHz clock, Plasticine has a peak floating-point performance of 12.3 single-precision Tflops and a total on-chip memory capacity of 16 MB, consuming a maximum power of 49 W. Plasticine provides an improvement of up to 76.9× in performance-per-watt over a conventional FPGA over a wide range of dense and sparse applications.

Acknowledgments

Graduate life at Stanford has truly been an exhilirating and rewarding experience. I have had the privilege to work with some of the best minds and visionaries in the field of computer system design, along with several knowledgeable, insightful, and brilliant colleagues who are now my friends and collaborators.

I would first like to thank Kunle Olukotun for his constant support, advice, and leadership throughout my graduate school. I have the highest respect for Kunle's accomplishments and his impact on the field, and he is the main reason I decided to come to Stanford. Kunle has been a role model and has led by example in identifying and tackling important, hard problems, and setting a high standard for research. During school, I have had the opportunity to engage in long discussions with Kunle regarding various open research questions. Not only did these discussions shape my own research in a positive way, it has influenced my thinking as a researcher and given me confidence to pursue hard problems with the required patience, discipline, and perseverance. Kunle sometimes likes saying in jest, "Remember: Your adviser is always right!". While I have tried my best to disprove that assertion during my graduate school, I must admit that my success has beeen limited.

I would also like to thank Christos Kozyrakis for being my mentor and guide, both academically and otherwise. Throughout graduate school, Christos has been encouraging of any new new research ideas and directions I had, and helped me boil them down to the core issues, no matter how absurd and far-fetched they sounded initially. The insights offered by Christos has helped me develop an objective mindset and ask new questions or consider new directions that I had not thought of myself, which has helped me better understand and explain my own research. Conversations with Christos have given me the strength to persevere when problems seemed insurmountable, without which I would not

have been able to pursue a PhD.

I have had the opportunity to interact with several faculty members who have helped me during the course of my research. I would like to thank Chris Re for serving on my reading committee and providing valuable feedback at several research retreats and during my thesis proposal. I would also like to thank Prof. Subhasish Mitra for serving on my orals committee, who I have had the privilege of interacting and collaborating with. Everything I know about digital design and VLSI is thanks to classes taught by him at Stanford. He also generously helped us with libraries and expertise to run various experiments for Plasticine. I would also like to thank John Hennessy for serving on my orals committee. I consider myself lucky to have had the opportunity to discuss my research with such an icon in the field of computer architecture.

I am fortunate to have worked in a healthy collaborative research environment with several fellow graduate student collaborators, who I have learnt a lot from. David Koeplinger is my fellow co-author, collaborator, and friend in all the published works that constitute this thesis. David's thoughtful ideas, calm mindset, and positive attitude have helped me navigate many roadblocks during the course of our research. Yaqi Zhang's systematic thinking and insights in several long discussions helped create the Plasticine architecture. Matt Feldman generously taught me juggling in additional to his technical insights. I also thank Stefan Hadjis for several insightful discussions. I extend my thanks to Tian Zhao, Ardavan Pedram, Ruben Fizel, Matt Vilim, Tushar Swamy, Luigi Nardi, Alex Rucker, Nathan Zhang, and Chris Aberger, from Kunle's group.

I also thank Mingyu Gao, Grant Ayers, Sam Grossman, Greg Kehoe, Ana Klimovic, Felipe Munera, and Adam Belay from Christos' group. I cherish the memorable experiences of building lab infrastructure for Stanford's undergraduate computer architecture course as a team with Grant, Sam, and Greg. I thank my fellow collaborators Jing Pu, Tony Wu, William Hwang, Artem Vasilyev, Xuan Yang from various other groups who helped in my research. I thank Tony Wu for his assistance with expertise and experiments for Chapter 4. I thank Jacob Bower from Maxeler Technologies for his help running experiments for Chapter 3.

Special thanks to Arvind Sujeeth, Kevin Brown, HyoukJoong Lee, Chris De Sa, Christina Delimitrou, and David Lo for their kind help and advice during my early years as a graduate

student.

I would like to thank my parents, brother, and sister-in-law for their encouragement and indefatigable support during my academic journey. Finally, I would like to thank my wife, Prakriti, for being my source of strength throughout the course of my PhD while braving the erratic schedules of an often preoccupied husband. She was instrumental in making graduate school the enjoyable experience that it was.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Rapid algorithmic and technological innovations in fields such as genome sequencing, data analytics, machine learning, and software-defined networking has placed greater compute and memory demands on the underlying computing systems. At the same time, technology scaling challenges with the slowdown of Moore's law and the end of Dennard scaling has made it increasingly difficult to scale processor performance in an area and energy-efficient manner. Consequently, the computer architecture community has ushered in the era of specialized accelerators [1–7]. Accelerators implement customized data and control paths to suit a domain of applications, thereby avoiding many of the overheads of flexibility in general-purpose processors. However, specialization in the form of dedicated ASICs is expensive due to the high NRE costs for design and fabrication, as well as the high deployment and iteration times. Furthermore, applications and algorithms evolve at a rapid pace; for example, the number of machine learning articles posted on arXiv.org has grown faster than Moore's law in the past decade [8]. An ASIC designed to accelerate a specific set of algorithms can immediately be rendered obsolete when ASIC design time is considerd along with the rate of algorithmic innovation [9]. This makes ASIC accelerators impractical for all but the most common and unchanging applications. Achieving a balance between specialization and flexibility in the underlying system is thus a critical task.

1.1 Flexibility: Instructions vs. Reconfigurable Datapaths

Flexibility in architectures can be achieved in multiple ways. Architectures such as general purpose CPUs and GPGPUs achieve flexibility by implementing a well-defined *Instruction Set Architecture* (ISA). Applications can be executed on such architectures by compiling them to a sequence of instructions in the ISA, which are then executed using one or more *threads*. ISA-based processors and thread-based execution models are ubiquitous today. However, instruction pipelines incur a nontrivial amount of hardware area and power overheads [10]; events such as instruction fetch, decode, and register file access account for about 40% of the datapath energy on the CPU [11], and about 30% [12] of the total dynamic power on the GPU. Furthermore, studies have shown that using a reconfigurable datapath in place of a conventional instruction pipeline in a GPU reduces energy consumption by about 57% [13]. Techniques like SIMD execution [14] amortize the overheads to some extent by performing more useful work per instruction, and can achieve energy efficiency improvements of 4% to 1.9x [15]. However, applications often contain parallelism at multiple levels of nesting [16]. ISAs typically offer limited support to exploit such nested parallelism even with SIMD, as costly synchronization mechanisms in software would be necessary to orchestrate execution. Architectures that allow a finer degree of customization can better exploit nested parallelism without incurring the overhead of instructions.

Reconfigurable architectures like Field Programmable Gate Arrays (FPGAs) achieve energy efficiency by providing statically reconfigurable compute elements and on-chip memories in a programmable interconnect that can be configured to implement customized datapaths. In FPGAs, these custom datapaths are configurable at the bit level, allowing users to prototype arbitrary digital logic and take advantage of architectural support for arbitrary precision computation. The performance and energy advantages of FPGAs are now motivating the integration of reconfigurable logic into data center computing infrastructures. FPGAs have already been used to deploy services commercially [17–21], and has been exposed as a rentable resource on the AWS F1 cloud [22]. However, bit-level reconfigurability in FPGA flexibility creates two key kinds of inefficiencies: *programming* and *architectural* inefficiencies.

1.2 Inefficiencies of Fine-Grained Reconfigurability

Programming inefficiencies are introduced due to low-level programming models using VHDL and Verilog, along with long compile times that take several hours. The inaccessibility of such programming models to most software developers has hindered widespread FPGA adoption for years [23]. Creating custom accelerator architectures on an FPGA is a complex task, requiring the coordination of large numbers of small, local memories, communication with off-chip memory, and the synchronization of many compute stages. Because of this complexity, attaining the best performance on FPGAs has traditionally required detailed hardware design using hardware description languages (HDL) like Verilog and VHDL. This low-level programming model has largely limited the creation of efficient custom hardware to experts in digital logic and hardware design.

In the past ten years, FPGA vendors and researchers have attempted to make reconfigurable logic more accessible to software programmers with the development of high-level synthesis (HLS) tools, designed to automatically infer register transaction level (RTL) specifications from higher level software programs. To better tailor these tools to software developers, HLS work has typically focused on imperative languages like C/C++, SystemC, and OpenCL [24]. Unfortunately, there are numerous challenges in inferring hardware from imperative programs. Imperative languages are inherently sequential and effectful. C programs in particular offer a number of challenges in alias analysis and detecting false dependencies [25], typically requiring numerous user annotations to help HLS tools discover parallelism and determine when various hardware structures can be used [26]. Achieving efficient hardware with HLS tools often requires an iterative process to determine which user annotations are necessary, especially for software developers less familiar with the intricacies of hardware design [27].

In addition to programming inefficiencies, *architectural inefficiencies* due to bit-level reconfigurability in computation and interconnect resources result in significant area and power overheads. For example, over 60% of the chip area and power in an FPGA is spent in the programmable interconnect [28–31]. In contrast, a study on an AMD GPU reports that up to 14% of the dynamic power is consumed in the interconnect [32]. Long combinational paths through multiple logic elements limit the maximum clock frequency at which

an accelerator design can operate. To mitigate these overheads, modern commercial FPGA architectures such as Intel's Arria 10 and Stratix 10 device families have evolved to include increasing numbers of coarse-grained blocks, including integer multiply-accumulators ("DSPs"), floating point units, pipelined interconnect, and DRAM memory controllers. The interconnect in these FPGAs, however, remains fine-grained to enable the devices to serve their original purpose as prototyping fabrics for arbitrary digital logic.

1.3 Mitigating Inefficiencies: Overlays and CGRAs

Several approaches have been proposed in the past to mitigate programming and architectural overheads of FPGAs. One approach that has been deployed frequently is the idea of *overlay architectures* [33]. An overlay architecture is a design that is implemented on top of an existing FPGA, using the underlying FPGA's fine-grained reconfigurable resources. Project Brainwave from Microsoft [19] is an example of a recent commercially deployed overlay architecture on an Altera Stratix X. Such architectures typically support a higher level programming model which do not require several hours of compile times. The overlay model also allows tuning and customizing the architecture to match specific needs of applications. However, overlay architectures do not mitigate FPGA hardware overheads, and hence achieve poor compute density.

To mitigate hardware overheads, several spatially reconfigurable architectures consisting of coarser-grained building blocks such as ALUs, register files, and memory controllers have been proposed. Such coarse-grained reconfigurable architectures (CGRAs) distribute processing blocks and memories in a programmable word-level static interconnect. CGRAs provide dense compute resources, power efficiency, and clock frequencies up to an order of magnitude higher than FPGAs. Several CGRAs have been proposed in the past decade [34–41]. Unfortunately, most of the proposed CGRAs still require low-level programming to extract high performance. The heterogeneity of resources in most CGRAs and in FPGAs with coarse-grain blocks adds further complications.

1.4 Co-Designing Hardware and Programming Model

Applications amenable to hardware acceleration typically exhibit certain key characteristics such as parallelism at multiple levels of nesting and data locality. The architecture must be able to exploit these characteristics with the right set of reconfigurable primitives to achieve high execution efficiency. The choice and granularity of reconfigurable primitives used in the architecture impacts its configuration interface, which, in turn, significantly impacts the complexity and efficiency of the compiler. Reconfigurable architectures designed in isolation without considering application characteristics or compiler flows can end up being imbalanced, with either too much flexibility and associated compiler complexity, or too little flexibility by hardwiring the architecture to a narrow domain of operations. Achieving the right balance between efficiency, flexibility, and programmability in reconfigurable architectures requires a full stack solution encomassing the programming model, compiler, and architecture.

Recent research has used domain-specific languages [42, 43] to capture rich semantic information from application programs that can be expressed using a set of domain-agnostic, high-level primitivies called *parallel patterns*. Parallel patterns such as map, reduce, filter, and flatmap have been successfully used to simplify parallel programming and code generation for a diverse set of parallel architectures including multi-core chips [44–46], GPUs [16, 47], and FPGAs [26].

This dissertation addresses both programming and architectural inefficiencies in FPGAs by co-designing the programming model, compiler flow, and the underlying hardware substrate while considering application characteristics. First, programming inefficiences are addressed using a set of composable hardware templates that capture parallelism and locality are proposed. The templates are designed to be automatically generated from high-level parallel patterns, as well as facilitate low-level optimizations such as resource estimation and design space exploration. The templates are targeted from parallel patterns from an experimental compiler using two key optimizations: *tiling* and *metapipelining* and evaluated using several benchmarks on an Altera Stratix V. Using the insights from the templates, hardware inefficiences are then addressed with a new coarse-grained reconfigurable architecture called Plasticine. Plasticine is a new spatially reconfigurable architecture designed

to efficiently execute applications composed of high-level parallel patterns. With an area footprint of $113mm^2$ in a 28-nm process and a 1-GHz clock, Plasticine has a peak floating point performance of 12.3 single-precision TFLOPS and a total on-chip memory capacity of 16 MB, consuming a maximum power of 49 W. Plasticine provides an improvement of up to $76.9\times$ in performance-per-watt over a conventional FPGA over a wide range of dense and sparse applications.

The rest of this thesis is organized as follows: Chapter 2 reviews key concepts in parallel patterns, high level synthesis, and motivates the requirements for a template-based hardware representation. Chapter 3 describes a set of composable hardware templates and a compiler flow to automatically generate hardware using these templates from high level parallel patterns. Chapter 4 describes and evaluates the Plasticine architecture based on the templates in 3. Chapter 5 discusses related work. Chapter 6 provides concluding remarks.

Chapter 2

Background

2.1 A Primer on Parallel Patterns

Parallel patterns are an extension to traditional functional programming which capture parallelizable computation on both dense and sparse data collections along with corresponding memory access patterns. Parallel patterns enable simple, automatic program parallelization rules for common computation tasks while also improving programmer productivity through higher level abstractions. The performance benefit from parallelization, coupled with improved programmer productivity, has caused parallel patterns to become increasingly popular in a variety of domains, including machine learning, graph processing, and database analytics [42, 43]. Parallel patterns have been efficiently mapped to hardware targets such as multicore [44–46], clusters [48–50], GPUs [16, 47], and FPGAs [26, 51].

2.1.1 Parallel Patterns Descriptions

As multi-core CPUs, knowledge of data parallelism is vital to achieve good performance when targeting FPGAs. This implicit knowledge makes parallel patterns a natural programming model to generate efficient FPGA designs. Our programming model is based on the parallel patterns *Map*, *FlatMap*, *Fold*, and *GroupByFold*. These patterns are selected because they are most amenable to hardware acceleration, and capture a wide variety of applications. Previous work [52–54] has shown how to stage a DSL application, lowering

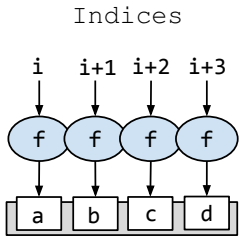
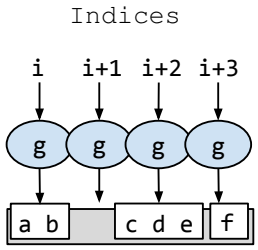
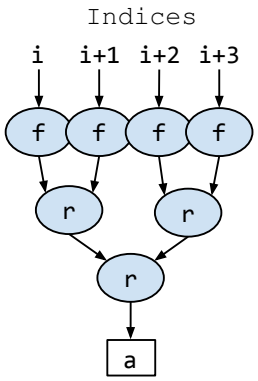
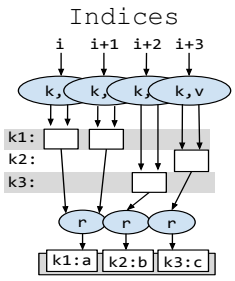
Map	FlatMap
	
Fold	**GroupByFold**
	

Table 2.1: The parallel patterns in our programming model.

it into a parallel pattern IR, as well as how to perform multiple high-level optimizations such as loop fusion automatically on the IR.

Table 2.1 depicts conceptual examples of each pattern, where computation is shown operating on four indices simultaneously. Every pattern takes as input one or more functions and an *index domain* describing the range of values that the pattern operates over.

We separate our parallel patterns into two groups, as shown in Figure 2.1. Multidimensional patterns have an arbitrary arity domain and range, but are restricted to a range which is a fixed function of the domain. One-dimensional patterns can have a dynamic output size. All patterns generate output values by applying a function to every index in the domain. Each pattern then merges these values into the final output in a different way. The output type V can be a scalar or structure of scalars. We denote multidimensional array types as V_R, which denotes a tensor of element type V and arity R. In Figure 2.1, subscript R always represents the arity of the output range, and D the arity of the input domain.

Parallel Pattern Definition		Code Example
Multidimensional		

`Map(d)(m)`	: V_D	`map(s){i => 2*x(i) }`
		
		`map(s){i => x(i) + y(i) }`

`MultiFold(d)(r)(z)(f)(c)`	: V_R	`multiFold(s)(1)(1){ i =>`
		`  (0, acc => acc + x(i))`
		`}{ (a,b) => a + b }`
		
		`multiFold(s,t)(r)(zeros(s)){ (i,j) =>`
		`  (i, acc => acc + x(i,j) )`
		`}{ (a,b) => map(s){ i => a(i) + b(i) } }`

One-dimensional		
`FlatMap(d)(n)`	: V_1	`flatMap(s){ i =>` `  if (x(i) > 0) [x(i)] else [] }`

`GroupByFold(d)(z)(g)(c)`	: $(K,V)_1$	`groupByFold(s)(0){ i =>`
		`  (x(i)/10, acc => acc + 1)`
		`}{ (a,b) => a + b }`

User-defined Values	
d : Integer_D	input domain
m : Index_D => V	value function
r : Integer_R	output range
n : Index => V_1	multi-value function
z : V_R	init accumulator
f : Index_D => (Index_R, V_R => V_R)	(location, value) function
c : (V_R,V_R) => V_R	combine accumulator
g : Index => (K, V => V)$_1$	(key, value) function

Figure 2.1: Definitions and usage examples of supported parallel patterns.

The parallel patterns in our programming model are described below.

Map creates a single output element per index using the function f, where each execution of f is guaranteed to be independent. The number of output elements from Map is the same as the size of the input iteration domain. Based on the number of collections read in f and the access patterns of each read, Map can capture the behavior of a gather, a standard element-wise map, a zip, a windowed filter, or any combination thereof. Note that the value function can close over an arbitrary number of input collections, and therefore this pattern can represent classic parallel operations like *map*, *zip*, and *zipWithIndex*.

FlatMap produces an arbitrary number of elements per index using function g, where again function execution is independent. The produced elements are concatenated into a flat output. Conditional data selection (e.g. *WHERE* in SQL, *filter* in Haskell or Scala) is a special case of FlatMap where g produces zero or one elements.

Fold first acts as a Map, producing a single element per index using the function f, then reduces these elements using an associative combine function r.

GroupByFold generates a hash key and a value for every index using functions k and v, respectively. Values with the same corresponding key are reduced on the fly into a single accumulator using an associative combine function r. GroupByFold may either be dense, where the space of keys is known ahead of time and all accumulators can be statically allocated, or sparse, where the pattern may generate an arbitrary number of keys at runtime. Histogram creation is a common, simple example of GroupByFold where the *key* function gives the histogram bin, the *value* function is defined to always be "1", and the *combine* function is integer addition.

MultiFold is a generalization of a *fold* which reduces generated values into a specified region of a (potentially) larger accumulator using an associative combine function. The initial value z is required to be an identity element of this function, and must have the same size and shape as the final output. The main function f generates an index specifying the location within the accumulator at which to reduce the generated value. We currently require the generated values to have the same arity as the full accumulator, but they may be of any size up to the size of the accumulator. Note that a traditional *fold* is the special case of MultiFold where every generated value is the full size of the accumulator. f then

```
1  val a: Matrix[Float]  // M x N
2  val b: Matrix[Float]  // N x P
3  val c = Map(M, P){(i,j) =>
4    // Outer Map function (f1)
5    Fold(N)(0.0f){k =>
6      // Inner map function (f2)
7      a(i,k) * b(k,j)
8    }{(x,y) =>
9      // Combine function (r)
10     x + y
11   }
12 }
```

Figure 2.2: Example of using Map and Fold in a Scala-based language for computing an untiled matrix multiplication using inner products.

converts each index into a function that consumes the specified slice of the current accumulator and returns the new slice. If the pattern's implementation maintains multiple partial accumulators in parallel, the combine function c reduces them into the final result.

2.1.2 Parallel Patterns Examples

Figure 2.2 shows an example of writing an untiled matrix multiplication with an explicit parallel pattern creation syntax. In this case, the Map creates an output matrix of size `M` $\times$ `P`. The Fold produces each element of this matrix using a dot product over `N` elements. Fold's map function (`f2`) accesses an element of matrix `a` and matrix `b` and multiplies them. Fold's combine function (`r`) defines how to combine arbitrary elements produced by `f2`, in this case using summation.

Figure 2.3 gives an example of using parallel patterns in a Scala-based language, where infix operators have been defined on collections which correspond to instantiations of parallel patterns. Note that in this example, the `filter` on line 3 creates a FlatMap with an index domain equal to the size of the `lineItems` collection. The `hashReduce` on line 5 creates a GroupByFold with an index domain with the size of the `before` collection.

Figure 2.4 implements the k-means clustering algorithm. k-means consumes a set of n sample points of dimensionality d and attempts to cluster those points by finding the k best cluster centroids for the samples. This is achieved by iteratively refining the centroid values. (Only one iteration of kmeans is shown in Figure 2.4 for simplicity.) First, every

```
1  val CUTOFF: Int = Date("1998-12-01")
2  val lineItems: Array[LineItem] = ...
3  val before = lineItems.filter{ item => item.date < CUTOFF }
4
5  val query = before.groupByFold { item =>
6    // Key function (k)
7    (item.returnFlag, item.lineStatus)
8  }{ item =>
9    // Value function (v)
10   val quantity = item.quantity
11   val price = item.extendedPrice
12   val discount = item.discount
13   val discountPrice = price * (1.0 - discount)
14   val charge = price * (1.0 - discount) * (1.0 + item.tax)
15   val count = 1
16   (quantity, price, discount, discountedPrice, count)
17 }{ (a,b) =>
18   // Combine function (r) - combine using summation
19   val quantity = a.quantity + b.quantity
20   val price = a.price + b.price
21   val discount = a.discount + b.discount
22   val discountPrice = a.discountPrice + b.discountPrice
23   val count = a.count + b.count
24   (quantity, price, discount, discountPrice, count)
25 }
```

Figure 2.3: Example of using filter (FlatMap) and GroupByFold in a Scala-based language, inspired by TPC-H query 1.

```
1  val points: Array[Array[Float]] = //data to be clustered, size n x d
2  val centroids: Array[Array[Float]] = // current centroids, size k x d
3
4  // Assign each point to the closest centroid by grouping
5  val groupedPoints = points.groupBy { pt1 =>
6    // Assign current point to the closest centroid
7    val minDistWithIndex = centroids.map { pt2 =>
8      pt1.zip(pt2).map { case (a,b) => square(a - b) }.sum
9    }.zipWithIndex.minBy(p => p._1)
10   minDistWithIndex._2
11 }
12
13 // Average of points assigned to each centroid
14 val newCentroids = groupedPoints.map { case (k,v) =>
15   v.reduce { (a,b) =>
16     a.zip(b).map { case (x,y) => x + y }
17   }.map { e => e / v.length }
18 }.toArray
```

Figure 2.4: k-means clustering implemented using Scala collections. In Scala, `_1` and `_2` refer to the first and second value contained within a tuple.

sample point is assigned to the closest current centroid by computing the distance between every sample and every centroid. Then new centroid values are computed by averaging all the samples assigned to each centroid. This process repeats until the centroid values stop changing.

One of the most important of these optimizations is fusing patterns together, both vertically (to decrease the reuse distance between producer-consumer relationships) and horizontally (to eliminate redundant traversals over the same domain). Figure 2.5 shows the structure of k-means after fusion rules have been applied. We have also converted the nested arrays in the Scala example to our multidimensional arrays. This translation requires the insertion of *slice* operations in certain locations, which produce a view of a subset of the underlying data.

2.2 The Need For A New Hardware Representation

Designing or generating efficient accelerator designs involves balancing compute with on-chip and off-chip memory bandwidth requirements to avoid resource bottlenecks. This

```
1  points: Array2D[Float](n,d) // data to be clustered
2  centroids: Array2D[Float](k,d) // current centroids
3
4  // Sum and number of points assigned to each centroid
5  (sums,counts) = multiFold(n)((k,d),k)(zeros((k,d),k)){ i =>
6    pt1 = points.slice(i, *)
7    // Assign current point to the closest centroid
8    minDistWithIndex = fold(k)((max, -1)){ j =>
9      pt2 = centroids.slice(j, *)
10     dist = fold(d)(0){ p =>
11       acc => acc + square(pt1(p) - pt2(p))
12     }{ (a,b) => a + b }
13     acc => if (acc._1 < dist) acc else (dist, j)
14   }{ (a,b) => if (a._1 < b._1) a else b }
15
16   minDistIndex = minDistWithIndex._2
17   sumFunc = ((minDistIndex, 0), acc => {
18     pt = points.slice(i, *)
19     map(d){ j => acc(j) + pt(j) }
20   })
21   countFunc = (minDistIndex, acc => acc + 1)
22
23   (sumFunc, countFunc)
24 }{ (a,b) => {
25   pt = map(k,d){ (i,j) => a._1(i,j) + b._1(i,j) }
26   count = map(k){ i => a._2(i) + b._2(i) }
27   (pt, count)
28 } }
29
30 // Average assigned points to compute new centroids
31 newCentroids = map(k,d){ (i,j) =>
32   sums(i,j) / counts(i)
33 }
```

Figure 2.5: k-means clustering represented using the parallel patterns in Figure 2.1 after fusion and code motion.

is irrespective of whether the accelerator is implemented as an ASIC or on an FPGA. In addition to managing data locality and exploiting nested parallelism as mentioned in Chapter 1, this process involves navigating a large multi-dimensional design space with application-level, architectural and microarchitectural parameters. The optimal set of parameters depends on the inherent parallelism and data locality in the application, as well as the available hardware resources. Heterogeneity in the underlying reconfigurable resources of FPGAs and CGRAs further complicates the design space. As a result, hardware accelerator design is an inherently iterative process which involves exploring a large design space for even moderately complex accelerators. Exhaustive or manual exploration of this space would be impractical for all but the simplest of designs, suggesting that efficient FPGA accelerator design requires support from high-level tools for rapid modeling and design space exploration. A promising approach is to construct a representation that internally represents hardware using general parameterized primitives. Such a representation can preserve locality and parallelism information in applications, as well as allow construction of target-specific resource estimators and design space exploration (DSE) tools to navigate the large search space.

High-level synthesis (HLS) tools such as LegUp [55] and Vivado HLS [56] (previously AutoPilot) [27] synthesize hardware from C. These tools provide estimates of the cycle count, area and power consumption along with hardware generation. However, imperative design descriptions place greater burden on the compiler to discover parallelism, pipeline structure and memory access patterns. The absence of explicit parallelism often leads to conservative compiler analyses producing sub-optimal designs. While some tools allow users to provide compiler hints in the form of directives or pragmas in the source code, this approach fails to capture key points in the design space. For example, consider Figure 2.6 which represents the gaussian discriminant analysis (GDA) kernel. All loops in this kernel are parallel loops. One set of valid design points would be to implement $L1$ as a coarse-grained pipeline with $L11$ and $L121$ as its stages. Commercial HLS tools support limited coarse-grained pipelining, but with serveral restrictions. For example, the *DATAFLOW* directive in Vivado HLS enables users to describe coarse-grained pipelines. However, the directive does not support arbitrarily nested coarse-grained pipelines, multiple producers and consumers between stages, or coarse-grain pipelining within a finite loop scope [57],

```
L1: for (int i=0; i<R; i++) {
  #pragma HLS PIPELINE II=1
  L11: for (int j=0; j<C; j++) {
    sub[j] = y[i] ? x[i][j]-mu0[j] : x[i][j]-mu1[j];
  }
  L121: for (int j1=0; j1<C; j1++) {
    L122: for (int j2=0; j2<C; j2++) {
      sigma[j1][j2] += sub[j1]*sub[j2];
  }
}
```

Figure 2.6: GDA for high-level synthesis.

as required in the outer loop in Figure 2.6.

In addition, compile times for HLS can be long for large designs due to the complications that arise during scheduling. Previous studies [58] point out other similar issues. Such limitations restrict the capability of HLS tools to explore more complex design spaces.

The next chapter describes hardware templates used in this work.

Chapter 3

A Template-Based Approach To Hardware Generation

3.1 Introduction

As mentioned in Chapter 1, the chief limiting factor in the general adoption of FPGAs is that their programming model is currently inaccessible to most software developers. High-level synthesis (HLS) compilers that produce Verilog from C/C++, SystemC, and OpenCL have limitations, as described in Chapter 2. Functional languages are a much more natural fit for high-level hardware generation as they have limited to no side effects and more naturally express a dataflow representation of applications which can be mapped directly to hardware pipelines [23]. Furthermore, the order of operations in functional languages is only defined by data dependencies rather than sequential statement order, exposing significant fine-grained parallelism that can be exploited efficiently in custom hardware.

Previous work [26, 51] has shown that compilers can utilize parallel patterns like the ones described in Section 2.1 to generate C- or OpenCL-based HLS programs and add certain annotations automatically. However, like hand-written HLS, the quality of the generated hardware is still highly variable. Apart from the practical advantage of building on existing tools, generating imperative code from a functional language only to have the HLS tool attempt to re-infer a functional representation of the program is a suboptimal solution because higher-level semantic knowledge in the original program is easily lost.

This chapter describes a template-based approach to generate hardware from parallel patterns. The requirements for hardware templates are motivated from the characteristics of parallel patterns, such that the underlying design space is fully captured. A series of compilation steps are descibed which automatically generate a low-level, efficient hardware design from an intermediate representation (IR) based on parallel patterns. Managing data locality and exploiting coarse-grained parallelism in applications are the two key challenges in generating efficient FPGA designs.

The rest of this chapter is organized as follows. Section 3.2 describes requirements for hardware templates from parallel patterns. Section 3.3 describes the list of compute and memory templates used to generate hardware. Section 3.4 describes our compiler flow to automatically generate hardware designs from parallel patterns. Section 3.5 evaluates our compiler flow using several benchmarks on an Altera Stratix V FPGA.

3.2 Hardware Implementation Requirements

Parallel patterns provide a concise set of parallel abstractions that can succinctly express a wide variety of machine learning and data analytic algorithms [42, 43, 47, 59]. By creating an architecture with specialized support for these patterns, we can execute these algorithms efficiently. This parallel pattern architecture requires several key hardware features, described below and summarized in Table 3.1.

First, all four patterns in Table 2.1 express data-parallel computation where operations on each index are entirely independent. An architecture with pipelined compute organized into SIMD lanes exploits this data parallelism to achieve a multi-element per cycle throughput. Additionally, apart from the lack of loop-carried dependencies, we see that functions f, g, k, and v in Table 2.1 are otherwise unrestricted. This means that the architecture's pipelined compute must be programmable in order to implement these functions.

Next, in order to make use of the high throughput available with pipelined SIMD lanes, the architecture must be able to deliver high on-chip memory bandwidth. In our programming model, intermediate values used within a function are typically scalars with statically known bit widths. These scalar values can be stored in small, distributed pipeline registers.

	Programming Model	Hardware
Compute	Parallel patterns	Pipelined compute SIMD lanes
On-Chip Memory	Intermediate scalars Tiled, linear accesses Random reads Streaming, linear accesses Nested patterns	Distributed pipeline registers Banked scratchpads Duplicated scratchpads Banked FIFOs Double buffering support
Off-Chip Memory	Linear accesses Random reads/writes	Burst commands Gather/scatter support
Interconnect	Fold FlatMap	Cross-lane reduction trees Cross-lane coalescing
Control	Pattern indices Nested patterns	Parallelizable counter chains Programmable control

Table 3.1: Programming model components and their corresponding hardware implementation requirements.

Collections are used to communicate data between parallel patterns. Architectural support for these collections depends on their associated memory access patterns, determined by analyzing the function used to compute the memory's address. For simplicity, we categorize access patterns as either statically predictable *linear* functions of the pattern indices or unpredictable, *random* accesses. Additionally, we label accesses as either *streaming*, where no data reuse occurs across a statically determinable number of function executions, or *tiled*, where reuse may occur. We use domain knowledge and compiler heuristics to determine if a random access may exhibit reuse.

Collections with tiled accesses can be stored in local scratchpads. To drive SIMD computation, these scratchpads should support multiple parallel address streams when possible. In the case of linear accesses, address streams can be created by banking. Parallel random reads can be supported by local memory duplication, while random write commands must be sequentialized and coalesced.

Although streaming accesses inevitably require going to main memory, the cost of main memory reads and writes can be minimized by coalescing memory commands and prefetching data with linear accesses. Local FIFOs in the architecture provide backing storage for both of these optimizations.

These local memories allow us to exploit locality in the application in order to minimize the number of costly loads or stores to main memory [60]. Reconfigurable banking support within these local memories increases the bandwidth available from these on-chip memories, thus allowing better utilization of the compute. Double buffering, generalized as N-buffering, support in scratchpads enables coarse-grain pipelined execution of imperfectly nested patterns.

The hardware design also requires efficient memory controllers to populate local memories and commit calculated results. As with on-chip memories, the memory controller should be specialized to different access patterns. Linear accesses correspond to DRAM burst commands, while random reads and writes in parallel patterns correspond to gathers and scatters, respectively.

Fold and FlatMap also suggest fine-grained communication across SIMD lanes. Fold requires reduction trees across lanes, while the concatenation in FlatMap is best supported by valid word coalescing hardware across lanes.

Finally, all parallel patterns have one or more associated loop indices. These indices can be implemented in hardware as parallelizable, programmable counter chains. Since parallel patterns can be arbitrarily nested, the architecture must also have programmable control logic to determine when each pattern is allowed to execute.

In the next section, we describe the Delite Hardware Definition Language (DHDL), which is composed of a set of parameterized hardware templates based on the requirements outlined in this section.

3.3 The Delite Hardware Definition Language

In this section, we describe the Delite Hardware Definition Language, or *DHDL*. DHDL is an intermediate language for describing hardware datapaths. A DHDL program describes a dataflow graph consisting of various kinds of nodes connected to each other by data dependencies. Each node in a DHDL program corresponds to one of the supported architectural templates listed in Table 3.2. DHDL is represented in-memory as a parameterized, hierarchical dataflow graph. A complete description of the capabilities of DHDL can be found in a published study [61].

A hardware datapath is described in DHDL using various nodes connected to each other by their data dependencies. DHDL also supports variable bit-width fixed-point types, variable precision floating point types, and associated type checking. Every node that either produces or stores data has an associated type. Table 3.2 describes the hardware templates and associated parameters supported in DHDL. There are four types of nodes:

Primitive Nodes

Primitive nodes correspond to basic operations, such as arithmetic and logic tasks, and multiplexers. Some complex multi-cycle operations such as *abs*, *sqrt* and *log* are also supported as primitive nodes. Every primitive node represents a vector computation; a "vector width" parameter defines the number of parallel instances of each node. Scalar operations are thus special cases where the associated vector width is 1.

Template		Description	Design Parameters
Primitive Nodes	+, -, *, /, <. >, mux	Basic arithmetic, logic, and control operations	Vector width, Type
	Ld, St	Load and store from on-chip memory	Vector width, Bank stride
Memories	OffChipMem	N-dimensional off-chip memory array	Dimensions, Type
	BRAM	On-chip scratchpad memory	Dimensions, Word width, Double buffering, Vector width, Banks, Int erleaving scheme, Type
	Priority Queue	Hardware sorting queue	Double buffering, Depth, Type
	Reg	Non-pipeline register	Double buffering, Vector width
Controllers	Counter	Counter chain used to produce loop iterators	Vector width
	Pipe	Hardware pipeline of primitive op erations. Typically used to represent bodies of innermost loops.	Parallelization factor, Pattern
	Sequential	Non-pipelined, sequential executi on of multiple stages.	Parallelization factor, Pattern
	Parallel	Fork-join style parallel containe r with synchronizing barrier.	
	MetaPipe	Coarse-grained pipeline with asynchronous handshaking signals across stages.	Parallelization factor, Pattern
Memory Command Generators	TileLd	Load a tile of data from an off-chip array	Tile dimensions, Word width, Parallelization factor
	TileSt	Store a tile of data to an off-chip array	Tile dimensions, Word width, Parallelization factor

Table 3.2: Description of templates in DHDL and supported parameters for each template

Memories

DHDL distinguishes between on-chip buffers and off-chip memory regions by representing them explicitly using separate nodes. This is used to capture on-chip and off-chip accesses which have different access times resource requirements. *OffChipMem* represents an N-dimensional region of memory stored on off-chip DRAM. *BRAM*, *Priority Queue* and *Reg* correspond to different types of on-chip buffers specialized for different kinds of computation. *OffChipMem*s are accessed using nodes called *memory command generators*, while on-chip buffers are accessed using primitive *Ld* (load) and *St* (store) nodes. The banking factor for a *BRAM* node is automatically calculated using the vector widths and access patterns of all the *Ld* and *St* nodes accessing it such that the required memory bandwidth can

be met.

Controllers

Several controller templates are supported in DHDL to capture imperfectly nested loops and parallelism at multiple nesting levels. Parallel patterns in input designs are represented using one of the *Pipe*, *MetaPipe*, or *Sequential* controllers with an associated *Counter* node. Each of these controllers is associated with a parallelization factor and the parallel pattern from which it was generated, which is used in replicating the nodes for parallelization. For example, nodes associated with the *map* pattern are replicated and connected in parallel, whereas nodes associated with the *reduce* pattern are replicated and connected as a balanced tree. *Pipe* is a dataflow pipeline which consists of purely primitive nodes. This typically represents innermost bodies of parallel loops that are traditionally converted to pipelines using software pipelining techniques. *MetaPipe* represents a coarse-grained pipeline where each of its stages are other controller nodes. *MetaPipe* orchestrates the execution of its stages in a pipelined fashion using asynchronous handshaking signals, thereby being able to tolerate variations in the execution times of each stage. Communication buffers used in between stages are converted to double buffers. *Sequential* represents unpipelined execution of a chain of controller nodes. *Parallel* is a container to execute multiple controller nodes in parallel with an implicit barrier at the end of execution. *Counter* is a simple chain of counters required to generate loop iterators. *Counter* has an associated vector width so that multiple successive iterators can be produced in parallel. This vector width is typically equal to the parallelization factor of the *Pipe*, *MetaPipe*, or *Sequential* it is associated with.

Memory Command Generators

*OffChipMem*s in DHDL are accessed at the granularity of *tiles*, where *tile* is an regular N-dimensional region of memory. Accesses to *OffChipMem*s are explicitly captured in DHDL using special *TileLd* (tile load) and *TileSt* (tile store) controllers. Each *TileLd* and *TileSt* node instantiates data and command queues to interface with the memory controller, and contains control logic to generate memory commands.

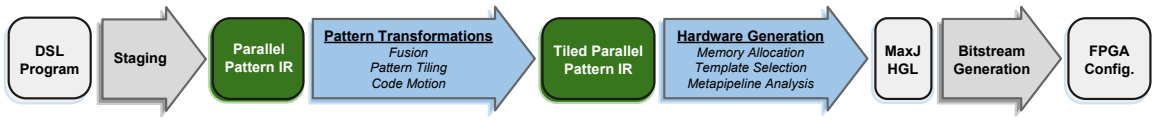


Figure 3.1: System diagram

3.4 Compiler Flow

This section describes our compiler flow to generate hardware automatically from parallel patterns using the DHDL templates described in Section 3.3. Figure 3.1 shows the system diagram. As seen in the figure, these steps fall into two categories: high level parallel pattern transformations (Section 3.4.1), and low level analyses and hardware generation optimizations (Section 3.4.2).

3.4.1 Parallel Pattern Transformations

One of the key challenges of generating efficient custom architectures from high level languages is in coping with arbitrarily large data structures. Since main memory accesses are expensive and FPGAs have a limited amount of fast local memory, our goal is to store a working set in the FPGA's local memory for as long as possible. Ideally, we also want to hide memory transfer latencies by overlapping communication with computation using prefetching hardware blocks.

Loop tiling has been extensively studied as a solution to this problem, as it allows data structures with predictable access patterns to be broken up into fixed size chunks. On FPGAs, these chunks can be stored locally in buffers. Tiling can also increase the reuse of these buffers by reordering computation, thus reducing the number of total accesses to main memory. Previous work on automated tiling transformations has focused almost exclusively on imperative C-like programs with only affine, data-independent memory access patterns. No unified procedure exists for automatically tiling a functional IR with parallel patterns.

This section outlines set of simple transformation rules which can be used to automatically tile parallel patterns. Because these rules rely on pattern matching rather than a mathematical model of the entire program, they can be used even on programs which

contain random and data-dependent accesses. We assume here that our input is a parallel pattern representation of a program and that well known target-agnostic transformations like fusion, code motion, and struct unwrapping have already been run.

The tiled intermediate representation exposes memory regions with high data locality, making them ideal candidates to be allocated on-chip. Certain semantic properties of memory regions, such as memory access pattern information, are preserved from parallel patterns. The memory access patterns are subsequently analyzed to automatically infer hardware structures like FIFOs, double buffers, and caches. We exploit parallelism at multiple levels by automatically inferring and generating *metapipelines*, hierarchical pipelines where each stage can itself be composed of pipelines and other parallel constructs. Our code generation approach involves mapping parallel IR constructs to a set of parameterizable hardware templates, where each template exploits a specific parallel pattern or memory access pattern. These hardware templates are implemented using a low-level Java-based hardware generation language (HGL) called MaxJ, from Maxeler Technologies [62].

We now describe our parallel pattern tiling transformations. Like classic loop tiling, our pattern tiling method is composed of two transformations: strip mining and interchange. The transformations described in this section are the result of a collaboration with other graduate students at Stanford, and is published in a study [59]. The following sections provide an intuition behind the transformations. A rigorous description of the transformation rules can be found in the aforementioned study.

Strip mining The strip mining algorithm is defined here using two passes over the IR. The first pass partitions each pattern's iteration domain d into tiles of size b by breaking the pattern into a pair of perfectly nested patterns. The outer pattern operates over the strided index domain, expressed here as d/b, while the inner pattern operates on a tile of size b. For the sake of brevity this notation ignores the case where b does not perfectly divide d. This case is trivially solved with the addition of *min* checks on the domain of the inner loop. In addition to splitting up the domain, patterns are transformed by recursively strip mining all functions within that pattern. Map is strip mined by reducing its domain and range and nesting it within a MultiFold. Note that the strided MultiFold writes to each memory location only once. In this case we indicate the MultiFold's combination function as unused

with an underscore. As defined in Figure 2.1, the MultiFold, GroupByFold, and FlatMap patterns have the property that a perfectly nested form of a single instance of one of these patterns is equivalent to a single "flattened" form of that same pattern. This property allows these patterns to be strip mined by breaking them up into a set of perfectly nested patterns of the same type as the original pattern.

The second strip mining pass converts array slices and accesses with statically predictable access patterns into slices and accesses of larger, explicitly defined array memory tiles. We define tiles which have a size statically known to fit on the FPGA using array copies. Copies generated during strip mining can then be used to infer buffers during hardware generation.

Pattern interchange Given an intermediate representation with strip mined nested parallel patterns, we now need to interchange patterns to increase the reuse of newly created data tiles. This can be achieved by moving strided patterns out of unstrided patterns. However, as with imperative loops, it is not sound to arbitrarily change the order of nested parallel patterns. We use two rules for pattern interchange adapted from a previously established *Collect-Reduce* reordering rule for computation on clusters [53]. These rules both match on the special case of MultiFold where every iteration updates the entire accumulator, which we refer to here as a *fold*. The first interchange rule defines how to move a scalar, strided *fold* out of an unstrided Map, transforming the nested loop into a strided *fold* of a Map. Note that this also changes the combination function of the *fold* into a Map. The second rule is the inverse of the first, allowing us to reorder a strided MultiFold with no reduction function (i.e. the outer pattern of a tiled Map) out of an unstrided *fold*. This creates a strided MultiFold of a scalar *fold*. We apply these two rules whenever possible to increase the reuse of tiled inputs.

Table 3.3 shows a simple example of the application of our pattern interchange rules on matrix multiplication. We assume here that code motion has been run again after pattern interchange has completed. In matrix multiplication, we interchange the perfectly nested strided MultiFold and the unstrided Map. This ordering increases the reuse of the copied tile of matrix y and changes the scalar reduction into a tile-wise reduction. Note that the partial result calculation and the inner reduction can now be vertically fused.

Transformation	Program After Transformation

High Level Language	`// Matrix Multiplication`
	`x: Array[Array[Float]] // m x p`
	`y: Array[Array[Float]] // p x n`
	`z = x.map{row =>`
	`  y.map{col =>`
	`  row.zipWith(col){(a,b) =>`
	`  a * b`
	`  }.sum`
	`  }`
	`}`

Strip Mined PPL	`multiFold(m/b0,n/b1)(m,n)(zeros(m,n)){(ii,jj) =>`
	`  ((ii,jj), zTile =>`
	`  map(b0,b1){(i,j) =>`
	`  tile = multiFold(p/b2)(1)(0){ kk =>`
	`  xTile = x.copy(b0 + ii, b2 + kk)`
	`  yTile = y.copy(b2 + kk, b1 + jj)`
	`  dprod = fold(b2)(0){ k =>`
	`  acc => acc + xTile(i,k) * yTile(k,j)`
	`  }{(a,b) => a + b})`
	`  (0, elemTile => elemTile + dprod)`
	`  }{(a,b) => a + b}`
	`  zTile(i,j) + tile`
	`  })`
	`}(_)`

Interchanged PPL	`multiFold(m/b0,n/b1)(m,n)(zeros(m,n)){(ii,jj) =>`
	`  tile = multiFold(p/b2)(b0,b1)(...){kk =>`
	`  xTile = x.copy(b0 + ii, b2 + kk)`
	`  yTile = y.copy(b2 + kk, b1 + jj)`
	`  (0, elemTile =>`
	`  map(b0,b1){(i,j) =>`
	`  dprod = fold(b2)(0){ k =>`
	`  acc => acc + xTile(i,j) * yTile(j,k)`
	`  }{(a,b) => a + b}`
	`  elemTile(i,j) + dprod`
	`  })`
	`  }{(a,b) =>`
	`  map(b0,b1){(i,j) => a(i,j) + b(i,j)`
	`  }`
	`  ((ii,jj), zTile =>`
	`  map(b0,b1){(i,j) => zTile(i,j) + tile(i,j)}`
	`  })`
	`}(_)`

Table 3.3: Example of the pattern interchange transformation applied to matrix multiplication.

```
1  (sums,counts) = multiFold(n/b0)((k,d),k)(...){ii =>
2    pt1Tile = points.copy(b0 + ii, *)
3    multiFold(b0)((k,d),k)(zeros(1,d),0){ i =>
4      pt1 = pt1Tile.slice(i, *)
5      minDistWithIndex = multiFold(k/b1)(1)((max, -1)){ jj =>
6        pt2Tile = centroids.copy(b1 + jj, *)
7        minIndTile = fold(b1)((max,-1)){ j =>
8          pt2 = pt2Tile.slice(j, *)
9          dist = distance(pt1, pt2)
10         acc => if (acc._1 < dist) acc else (dist, j+jj)
11       }{ (a,b) => if (a._1 < b._1) a else b }
12
13       (0, acc =>
14         if (acc._1 < minIndTile._1) acc else minIndTile)
15     }{(a,b) =>
16       if (a._1 < b._1) a else b
17     }
18
19
20     minDistIndex = minDistWithIndex._2
21     sumFunc = ... // Fig 4: lines 17-20
22     countFunc = ... // Fig 4: line 21
23     (sumFunc, countFunc)
24   }{(a,b) => ... /* Tiled combination function */ }
25   (0, acc => ... /* Tiled combination function */ )
26 }{(a,b) => ... /* Tiled combination function */ }
27
28 newCentroids = multiFold(k/b1,d)(k,d)(...){ (ii,jj) =>
29   sumsBlk = sums.copy(b1 + ii, *)
30   countsBlk = counts.copy(b1 + ii)
31   (ii, acc => map(k,d){ (i,j) =>
32     sumsBlk(i,j) / countsBlk(i)
33   })
34 }
```

Strip mined k-means in PPL.

Figure 3.2: Strip-mined k-means starting from the fused representation in Figure 2.5, using tile sizes of b_0 and b_1 for the number of points n and the number of clusters k. The number of features d is not tiled in this example.

	Fused	**Strip Mined**	**Interchanged**
points	$n \times d$	$n \times d$	$n \times d$
centroids	$n \times k \times d$	$n \times k \times d$	$(n/b_0) \times k \times d$
minDistWithIndex	0	0	0

Table 3.4: Comparison of off-chip reads for key k-means data structures.

```
1  (sums,counts) = multiFold(n/b0)((k,d),k)(...){ ii =>
2    pt1Tile = points.copy(b0 + ii, *)
3    minDistWithInds = multiFold(k/b1)(b1)(map(b1)((max, -1))){ jj =>
4      pt2Tile = centroids.copy(b1 + jj, *)
5      minIndsTile = map(b0){ i =>
6        pt1 = pt1Tile.slice(i, *)
7        minIndTile = fold(b1)((max,-1)){ j =>
8          pt2 = pt2Tile.slice(j, *)
9          dist = distance(pt1, pt2)
10         acc => if (acc._1 < dist) acc else (dist, j+jj)
11       }{ (a,b) => if (a._1 < b._1) a else b }
12     }
13     (0, acc => map(b0){ i =>
14       if (acc(i)._1 < minIndsTile(i)._1) acc else minIndsTile(i) })
15   }{(a,b) =>
16     map(b0){i => if (a(i)._1 < b(i)._1) a(i) else b(i) }
17   }
18   multiFold(b0)(k,d)(zeros(k,d)){ i =>
19     pt1 = pt1Tile.slice(i, *)
20     minDistIndex = minDistWithInds(i)._2
21     sumFunc = ... // Fig 4: lines 17-20
22     countFunc = ... // Fig 4: line 21
23     (sumFunc, countFunc)
24   }{(a,b) => ... /* Tiled combination function */ }
25   (0, acc => ... /* Tiled combination function */ )
26 }{(a,b) => ... /* Tiled combination function */ }
27
28 newCentroids = multiFold(k/b1,d)(k,d)(...){ (ii,jj) =>
29   sumsBlk = sums.copy(b1 + ii, *)
30   countsBlk = counts.copy(b1 + ii)
31   (ii, acc => map(k,d){ (i,j) =>
32     sumsBlk(i,j) / countsBlk(i)
33   })
34 }
```

Pattern Interchanged k-means in PPL.

Figure 3.3: k-means after pattern interchange transformation has been performed on Figure 3.2

	Fused	**Strip Mined**	**Interchanged**
points	d	$b_0 \times d$	$b_0 \times d$
centroids	d	$b_1 \times d$	$b_1 \times d$
minDistWithIndex	2	2	$2 \times b_0$

Table 3.5: Comparison of on-chip reads for key k-means data structures.

```
1      For each tile of b0 points:
2        Copy the points tile into local memory
3 - 4    For each point pt1 in the points tile:
5          For each tile of b1 centroids:
6            Copy the centroids tile into local memory
7 - 8        For each centroid pt2 in the centroids tile:
9              Compute distance between pt1 and pt2
10-11          Keep the closest (index,distance) pair
             End
13-16        Keep the closest pair across tiles
           End
20         Extract the index of the closest centroid
21         Add pt1 to row minDistIndex
22         Increment count at minDistIndex
24         Add point and count sums across tiles
         End
25-26    Add point and count sums across tiles
       End

28     For each tile of b1 point sums and counts:
29       Copy the point sums tile into local memory
30       Copy the point counts tile into local memory
31-32    Compute each new centroid as sums(i) / count(i)
       End
```

Figure 3.4: Textual description of strip mining example in Figure 3.2

```
1      For each tile of b0 points:
2        Copy the points tile into local memory
3        For each tile of b1 centroids:
4          Copy the centroids tile into local memory
5 - 6      For each point pt1 in the points tile:
7 - 8        For each centroid pt2 in the centroids tile:
9              Compute distance between pt1 and pt2
10-11          Keep the closest (index,distance) pair
             End
           End
13-16      For each point: keep the closest pair across tiles
         End
18-19    For each point pt1 in points tile:
20         Extract the index of the closest centroid
21         Add pt1 to row minDistIndex
22         Increment count at minDistIndex
24         Add point and count sums across tiles
         End
25-26    Add point and count sums across tiles
       End

28     For each tile of b1 point sums and counts:
29       Copy the point sums tile into local memory
30       Copy the point counts tile into local memory
31-32    Compute each new centroid as sum / count
       End
```

Figure 3.5: Textual description of patterm interchange example in Figure 3.3

Discussion The rules we outline here for automatic tiling of parallel patterns are target-agnostic. However, tile copies should only be made explicit for devices with scratchpad memory. Architectures with hierarchical memory systems effectively maintain views of subsections of memory automatically through caching, making explicit copies on these architectures a waste of both compute cycles and memory.

The framework presented in this chapter requires the user to explicitly specify tile sizes for all dimensions which require tiling. Other studies [61] have explored automatically selecting tile sizes in the compiler using modeling and design space exploration.

Example We conclude this section with a complete example of tiling the k-means clustering algorithm, starting from the fused representation shown in Figure 2.5. We assume here that we wish to tile the number of input points, n, with tile size b_0 and the number of clusters, k, with tile size b_1 but not the number of dimensions, d. This is representative of machine learning classification problems where the number of input points and number of labels is large, but the number of features for each point is relatively small.

Figures 3.2 and 3.3 shows a code representation of the k-means clustering algorithm after strip mining and after pattern interchange, respectively. A pseudocode representation of the strip-mined code in 3.2 is shown in Figure 3.4. Pseudocode for the code after pattern interchange is shown in Figure 3.5. Table 3.4 shows how the transformations affect the number of reads from off-chip memory in terms of elements, and Table 3.5 shows the number of on-chip reads. During strip mining, we create tiles for both the *points* and *centroids* arrays, which helps us to take advantage of main memory burst reads. However, in the strip mined version, we still fully calculate the closest centroid for each point. This requires the entirety of *centroids* to be read for each point. We increase the reuse of each tile of *centroids* by first splitting the calculation of the closest centroid label from the MultiFold (Figure 3.2. line 5). The iteration over the centroids tile is then perfectly nested within the iteration over the points. Interchanging these two iterations allows us to reuse the centroids tile across points, thus decreasing the total number of main memory reads for this array by a factor of b_0. This decrease comes at the expense of changing the intermediate (distance, label) pair for a single point to a set of intermediate pairs for an entire tile of *points*. Since the created intermediate result has size $2b_0$, we statically determine that this

is an advantageous tradeoff and use the split and interchanged form of the algorithm.

3.4.2 Hardware Generation

In this section, we describe how the tiled intermediate representation is translated into an efficient FPGA design. FPGAs are composed of various logic, register, and memory resources. These resources are typically configured for a specific hardware design using a hardware description language (HDL) that is translated into an FPGA configuration file. Our approach to FPGA hardware generation translates our parallel pattern IR into the DHDL templates described in Section 3.3. The templates are implemented in MaxJ, a Java-based hardware generation language (HGL), which is in turn used to generate an HDL. This is simpler than generating HDL directly because MaxJ performs tasks such as automatic pipelining of innermost loops and other low-level hardware optimizations.

We analyze the structure of the parallel patterns in the IR to determine the correct template to translate the pattern to hardware. Table 3.2 lists the hardware templates. Each template can be composed with other templates. For example, a *MetaPipe controller* could be composed of multiple *Parallel controller*s, each of which could contain other instances of controllers, or primitive nodes operating in SIMD or as a reduction tree. We next describe the key features in the IR which we use to infer each of these template classes.

Memory Allocation Generating efficient FPGA hardware requires effective usage of on-chip memories (buffers). Prior to generating MaxJ, we run an analysis pass to allocate buffers for arrays based on data access patterns and size. All arrays with statically known sizes, such as array *copies* generated in the tiling transformation described in Section 3.4.1, are assigned to buffers. Dynamically sized arrays are kept in main memory and we generate caches for any non-affine accesses to these arrays. We also track each memory's readers and writers and use this information to instantiate a template with the appropriate word width and number of ports.

Pipeline Execution Units We generate parallelized and pipelined hardware when parallel patterns compute with scalar values, as occurs for the innermost patterns. We implemented templates for each pipelined execution unit in Table 3.2 using MaxJ language

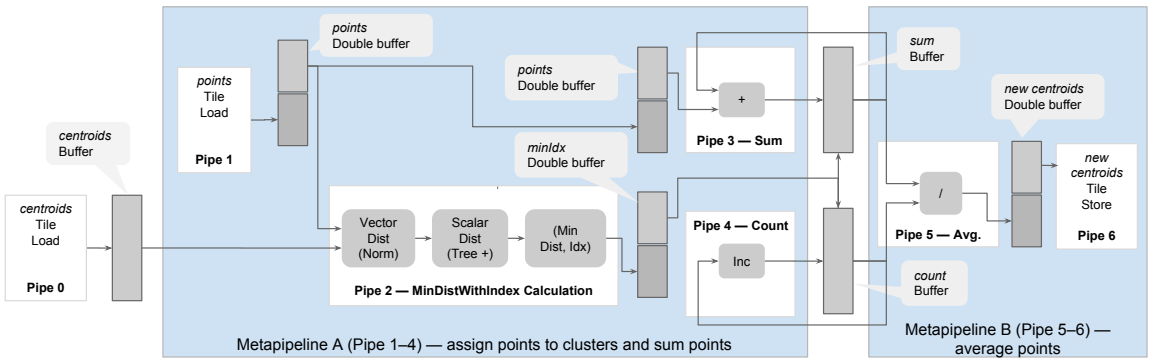


Figure 3.6: Hardware generated for the k-means application.

constructs, and instantiate each template with the proper parameters (e.g., data type, vector length) associated with the parallel pattern. The MaxJ compiler applies low-level hardware optimizations such as vectorization, code scheduling, and fine-grained pipelining, and generates efficient hardware. For example, we instantiate a reduction tree for a MultiFold over an array of scalar values, which is automatically pipelined by the MaxJ compiler.

Metapipelining To generate high performance hardware from parallel patterns, it is insufficient to exploit only a single level of parallelism. However, exploiting nested parallelism requires mechanisms to orchestrate the flow of data through multiple pipeline stages while also exploiting parallelism at each stage of execution, creating a hierarchy of pipelines, or *metapipeline*. This is in contrast to traditional HLS tools which require inner patterns to have a static size and be completely unrolled in order to generate a flat pipeline containing both the inner and outer patterns.

We create metapipeline schedules by first performing a topological sort on the IR of the body of the current parallel pattern. The result is a list of stages, where each stage contains a list of patterns which can be run concurrently. Exploiting the pattern's semantic information, we then optimize the metapipeline schedule by removing unnecessary memory transfers and redundant computations. For instance, if the output memory region of the pattern has been assigned to a buffer, we do not generate unnecessary writes to main memory.

As another example, our functional representation of tiled parallel patterns can sometimes create redundant accumulation functions, e.g., in cases where a MultiFold is tiled into a nested MultiFold. During scheduling we identify this redundancy and emit a single copy of the accumulator, removing the unnecessary intermediate buffer. Finally, in cases where the accumulator of a MultiFold cannot completely fit on-chip, we add a special forwarding path between the stages containing the accumulator. This optimization avoids redundant writes to memory and reuses the current tile. Once we have a final schedule for the metapipeline, we promote every output buffer in each stage to a double buffer to avoid write after read (WAR) hazards between metapipeline stages.

Example Figure 3.6 shows a block diagram of the hardware generated for the k-means application. For simplicity, this diagram shows the case where the *centroids* array completely fits on-chip, meaning we do not tile either the number of clusters k or the number of features d. The generated hardware contains three sequential steps. The first step (Pipe 0) preloads the entire *centroids* array into a buffer. The second step (MetaPipe A) is a metapipeline which consists of three stages with double buffers to manage communication between the stages. These three stages directly correspond to the three main sections of the MultiFold (Figure 2.5, line 5) used to sum and count the input points as grouped by their closest centroid. The first stage (Pipe 1) loads a tile of the *points* array onto the FPGA. Note that this stage is double buffered to enable hardware prefetching. The second stage (Pipe 2) computes the index of the closest centroid using vector compute blocks and a scalar reduction tree. The third stage (Pipe 3 and Pipe 4) increments the count for this minimum index and adds the current point to the corresponding location in the buffer allocated for the *new centroids*. The third step (MetaPipe B) corresponds with the second outermost parallel pattern in the k-means application. This step streams through the point sums and the centroid counts, dividing each sum by its corresponding count. The resulting new centroids are then written back to main memory using a tile store unit for further use on the CPU.

Our automatically generated hardware design for the core computation of k-means is very similar to the manually optimized design described by Hussain et al. [63]. While the manual implementation assumes a fixed number of clusters and a small input dataset

which can be preloaded onto the FPGA, we use tiling to automatically generate buffers and tile load units to handle arbitrarily sized data. Like the manual implementation, we automatically parallelize across centroids and vectorize the point distance calculations. As we see from the k-means example, our approach enables us to automatically generate high quality hardware implementations which are comparable to manual designs.

3.5 Evaluation

We evaluate our approach to hardware generation described in Section 3.4 by comparing the performance and area utilization of the FPGA implementations of a set of data analytic benchmarks. We focus our investigation on the relative improvements that tiling and metapipelining provide over hardware designs that do not have these features.

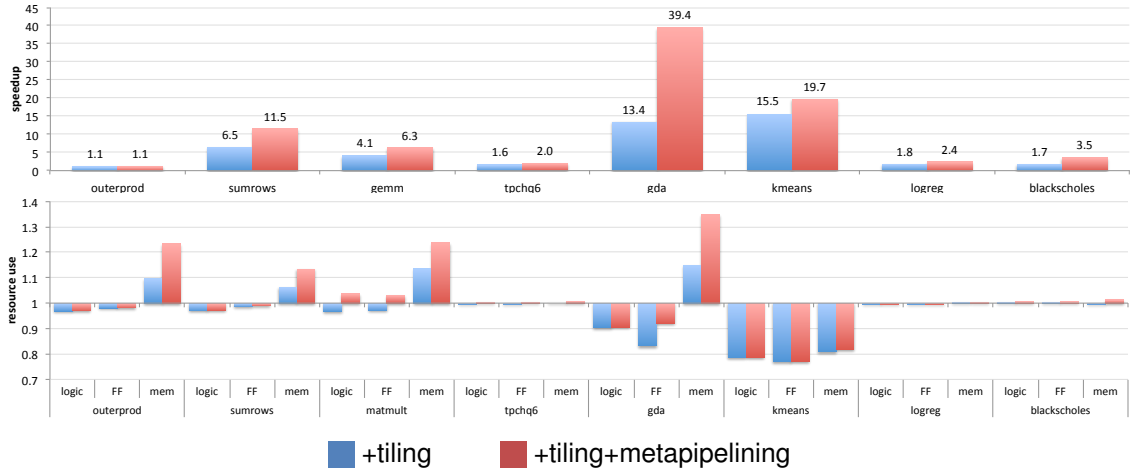


Figure 3.7: Speedups and resource usages, relative to respective baseline designs, resulting from tiling and metapipelining.

3.5.1 Methodology

The benchmarks used in our evaluation are summarized in Table 3.6. We choose to study vector outer product, matrix row summation, and matrix multiplication as these exemplify many commonly occurring access patterns in the machine learning domain. TPC-H Query

Benchmark	Description	Collections Ops
outerprod	Vector outer product	*map*
sumrows	Summation through matrix rows	*map, reduce*
gemm	Matrix multiplication	*map, reduce*
tpchq6	TPC-H Query 6	*filter, reduce*
logreg	Logistic regression	*map, reduce*
gda	Gaussian discriminant analysis	*map, filter, reduce*
blackscholes	Black-Scholes option pricing	*map*
kmeans	k-means clustering	*map, groupBy, reduce*

Table 3.6: Evaluation benchmarks with major collections operations used by Scala implementation.

6 is a data querying application which reads a table of purchase records, filtering all records which match a given predicate. It then computes the sum of a product of two columns in the filtered records. Logistic regression is a binary discriminative classification algorithm that uses the sigmoid function in the calculation of predictions. Gaussian discriminant analysis (GDA) is a classification algorithm which models the distribution of each class as a multivariate Gaussian. Black-Scholes is a financial analytics application for option pricing. k-means clustering groups a set of input points by iteratively calculating the k best cluster centroids. In our implementations, all of these benchmarks operate on single precision, floating point data.

We implement our transformation and hardware generation steps in an existing compiler framework called Delite [46]. We write each of our benchmark applications in OptiML [64], a high level, domain specific language embedded in Scala for machine learning. We then compile each of these applications with the modified Delite compiler. During compilation, applications are staged, translating them into PPL representations. The compiler then performs the tiling transformations and hardware optimizations described in Sections 3.4.1 and 3.4.2 before generating the DHDL hardware templates implemented in MaxJ. We then use the Maxeler MaxCompiler toolchain to generate an FPGA configuration bitstream from our generated MaxJ. We use the Maxeler runtime layer to manage communication with the FPGA from the host CPU. We measure the running times of these designs starting

after input data has been copied to the FPGA's DRAM and ending when the hardware design reports completion. Final running times were calculated as an arithmetic mean of five individual run times to account for small runtime variations in main memory accesses and Maxeler's device driver stack.

We run each generated design on an Altera Stratix V FPGA on a Max4 Maia board. The Maia board contains 48 GB of DDR3 DRAM with a maximum bandwidth of 76.8 GB/s. The area numbers given in this section are obtained from synthesis reports provided by Altera's logic synthesis toolchain. Area utilization is reported under three categories: Logic utilization (denoted "logic"), flip flop usage ("FF"), and on-chip memory usage ("mem").

3.5.2 Experiments

The baseline for each benchmark is an optimized hardware design implemented using MaxJ. The baseline designs were manually tuned after automatic generation and are representative of optimizations done by state-of-the-art high-level synthesis tools. In particular, each baseline design exploits data and pipelined parallelism within patterns where possible. Pipelined parallelism is exploited for patterns that operate on scalars. Our baseline design exploits locality at the level of a single DRAM burst, which on the MAX4 MAIA board is 384 bytes. To isolate the effects of the amount of parallelism in our comparison, we keep the innermost pattern parallelism factor constant between the baseline design and our optimized versions for each benchmark.

We evaluate our approach against the baseline by generating two hardware configurations per benchmark: a configuration with tiling but no metapipelining, and a configuration with both tiling and metapipelining optimizations enabled.

Impact of tiling alone Figure 3.7 shows the obtained speedups as well as relative on-chip resource utilizations for each of benchmarks. As can be seen, most benchmarks in our suite show significant speedup when tiling transformations are enabled. Benchmarks like *sumrows* and *gemm* benefit from inherent locality in their memory accesses. For *gemm*, our automatically generated code achieves a speedup of $4\times$ speedup over the baseline for a marginal increase of about 10% on-chip memory usage.

Benchmarks *outerprod* and *tpchq6* do not show a significant difference with our tiling transformations over the baseline. This is because both *outerprod* and *tpchq6* are both memory-bound benchmarks. *Tpchq6* streams through the input once without reuse, and streaming data input is already exploited in our baseline design. *Blackscholes* has a similar data access pattern as *tpchq6*, due to which it achieves a speedup similar to that of *tpchq6*. Hence tiling does not provide any additional benefit. Most of the locality in *logreg* is already captured at burst-level granularity by our baseline. As a result, *logreg* achieves a modest speedup of $1.8x$ over the baseline due to tiling. The core compute pipeline in *outerprod* is memory-bound at the stage writing results to DRAM, which cannot be addressed using tiling. Despite the futility of tiling in terms of performance, tiling *outerprod* has a noticeable increase in memory utilization as the intermediate result varies as the square of the tile size.

In *kmeans* and *gda*, some of the input data structures are small enough that they can be held in on-chip memory. This completely eliminates accesses to off-chip memory, leading to dramatic speedups of $13.4\times$ and $15.5\times$ respectively with our tiling transformations. *gda* uses more on-chip memory to store intermediate data. On the other hand, the tiled version *kmeans* utilizes less on-chip memory resources. This is because the baseline for *kmeans* instantiates multiple load and store units, each of which creates several control structures in order to read and write data from DRAM. Each of these control structures includes address and data streams, which require several on-chip buffers. By tiling, we require a smaller number of load and store units.

Impact of metapipelining The second speedup bar in Figure 3.7 shows the benefits of metapipelining. Metapipelines increase throughput at the expense of additional on-chip memory used for double buffers. Metapipelining overlaps design compute with data transfer and helps to hide the cost of the slower stage. Benchmarks like *gemm* and *sumrows* naturally benefit from metapipelining because the memory transfer time is completely overlapped with the compute, resulting in speedups of $6.3\times$ and $11.5\times$ respectively. Metapipelining also exploits overlap in streaming benchmarks like *tpchq6* and *blackscholes*, where the input data is fetched and stored simultaneously with the core computation.

The speedup due to metapipelining is largely determined by balancing between stages.

Stages with roughly equal number of cycles benefit the most, as this achieves the most overlap. Unbalanced stages are limited by the slowest stage, thus limiting performance. We observe this behavior in *outerprod*, where the metapipeline is bottlenecked by the stage writing results back to DRAM. The metapipeline in *logreg* is bottlenecked at the stage performing dot products of all the points in the input tile with the *theta* vector. As we only parallelize the innermost parallel pattern in this work, only a single dot product is produced at a time, even though the dot product itself is executed in parallel across the point dimensions. On the other hand, applications like *gda*, *kmeans* and *sumrows* greatly benefit from metapipelining. In particular, *gda* naturally maps to nested metapipelines that are well-balanced. The stage loading the input tile overlaps execution with the stage computing the output tile and the stage storing the output tile. The stage computing the output tile is also a metapipeline where the stages perform vector subtraction, vector outer product and accumulation. We parallelize the vector outer product stage as it is the most compute-heavy part of the algorithm; parallelizing the vector outer product enables the metapipeline to achieve greater throughput. This yields an overall speedup of $39.4\times$ over the baseline.

3.6 Conclusion

In this chapter, we introduced a set of composable, parameterized hardware templates to generate efficient FPGA hardware. We introduced a set of compilation steps necessary to produce an efficient FPGA hardware design from an intermediate representation composed of nested parallel patterns. We described a set of simple transformation rules which can be used to automatically tile parallel patterns which exploit semantic information inherent within these patterns and which place fewer restrictions on the program's memory accesses than previous work. We then presented a set of analysis and generation steps which can be used to automatically infer optimized hardware designs with metapipelining. Finally, we presented experimental results for a set of benchmarks in the machine learning and data querying domains which show that these compilation steps provide performance improvements of up to $39.4\times$ with a minimal impact on FPGA resource usage.

Chapter 4

Plasticine: A Reconfigurable Architecture For Parallel Patterns

The previous chapter addressed FPGA programmability inefficiencies identified in Chapter 1 by describing a flow to produce hardware designs from parallel patterns, thereby raising the level of abstraction for programmers. This chapter addresses the architectural inefficiencies of FPGAs by describing Plasticine, a new reconfigurable architecture designed to efficiently execute applications composed of parallel patterns.

4.1 Introduction

Several studies have shown that bit-level reconfigurability in FPGAs incurs significant area and power overheads [28–31], most notably in the interconnect. FPGAs are built with the capability to implement arbitrary digital designs. Doing so requires the ability to support building arbitrary chains of combinational and sequential logic, which need to be placed, routed, and timed on a large array containing hundreds of thousands of reconfigurable elements. This degree of flexibility allowed in the architecture, coupled with generic, low-level programming models, creates an enormous mapping search space within the compiler with several constraints to be satisfied. Navigating this space and producing an efficient design, as a result, takes several hours.

Various types of coarse-grained building blocks have been proposed to counteract some

of the overheads of fine-grained reconfigurability. Bus-based connections in FPGA interconnects have been shown to reduce routing area by up to 14% [65]. Coarse-grained units like floating point multipliers and DDR controllers are making their way FPGA architectures [66]. Modern FPGAs also include registers in the static switches [67] to reduce long combinational paths through the interconnect, which also facilitates using a higher fabric clock frequency. While these measures improve overall compute density on chip, they also increase the heterogeneity of resources which further complicates compilation. As a result, both fine-grained FPGAs and many CGRAs still require low level programming and suffer from long compilation times.

This work focuses on developing a coarse-grain, reconfigurable fabric with direct architectural support for parallel patterns which is both highly efficient in terms of area, power, and performance and easy to use in terms of programming and compilation complexity. Building on the insights from Chapter 3, this chapter introduces *Plasticine*, a new spatially reconfigurable architecture designed to efficiently execute applications composed of parallel patterns.

We motivate Plasticine using key application characteristics captured by parallel patterns that are amenable to hardware acceleration, such as hierarchical parallelism, data locality, memory access patterns, and control flow, described earlier in section 3.2. Based on these observations, we architect Plasticine as a collection of *Pattern Compute Units* (PCUs) and *Pattern Memory Units* (PMUs). PCUs are multi-stage pipelines of reconfigurable SIMD functional units that can efficiently execute nested patterns. PCUs contain hardware support for cross-SIMD lane shifting and reduction operations. Data locality is exploited in PMUs using banked scratchpad memories and configurable address decoders. Scratchpads in the PMU are configurable to support streaming and double buffered accesses, which are required to implement *metapipelines* described in Chapter 3 and exploit coarse-grained pipeline parallelism. Multiple on-chip Address Generators (AG) and scatter-gather engines in Coalescing Units (CU) make efficient use of DRAM bandwidth by supporting a large number of outstanding memory requests, memory coalescing, and burst mode for dense accesses. These units communicate with each other through a pipelined, statically configured interconnect with separate bus-level and word-level data, and bit-level

control networks. Hierarchy in Plasticine's datapath simplifies compiler mapping and improves execution efficiency; the compiler can map inner loop bodies to PCUs such that most operands are transferred directly between functional units without scratchpad accesses or inter-PCU communication.

Plasticine is implemented in RTL using Chisel, a Scala-based hardware description language. Plasticine has an area footprint of 113 mm^2 in a 28nm process, and consumes a maximum power of 49 W at a 1 GHz clock. Using VCS and DRAMSim2 for cycle-accurate simulation, we evaluate Plasticine on a wide range of dense and sparse data benchmarks in the domains of linear algebra, machine learning, data analytics and graph analytics. We demonstrate that Plasticine provides an improvement of up to 76.9$\times$ in performance-per-Watt over a conventional FPGA.

Section 4.2 introduces the Plasticine architecture and explores key design tradeoffs. Section 4.3 evaluates the power and performance efficiency of Plasticine versus an FPGA.

4.2 The Plasticine Architecture

Plasticine is a tiled architecture consisting of reconfigurable *Pattern Compute Units* (PCUs) and *Pattern Memory Units* (PMUs), which we refer to collectively simply as "units". Units communicate with three kinds of static interconnect: word-level scalar, multiple-word-level vector, and bit-level control interconnects. Plasticine's array of units interfaces with DRAM through multiple DDR channels. Each channel has an associated address management unit that arbitrates between multiple address streams, and consists of buffers to support multiple outstanding memory requests and address coalescing to minimize DRAM accesses. Each Plasticine component is used to map specific parts of applications: local address calculation is done in PMUs, DRAM address computation happens in the DRAM address management units, and the remaining data computation happens in PCUs. Note that the Plasticine architecture is parameterized; we discuss the sizing of these parameters in Section 4.2.7

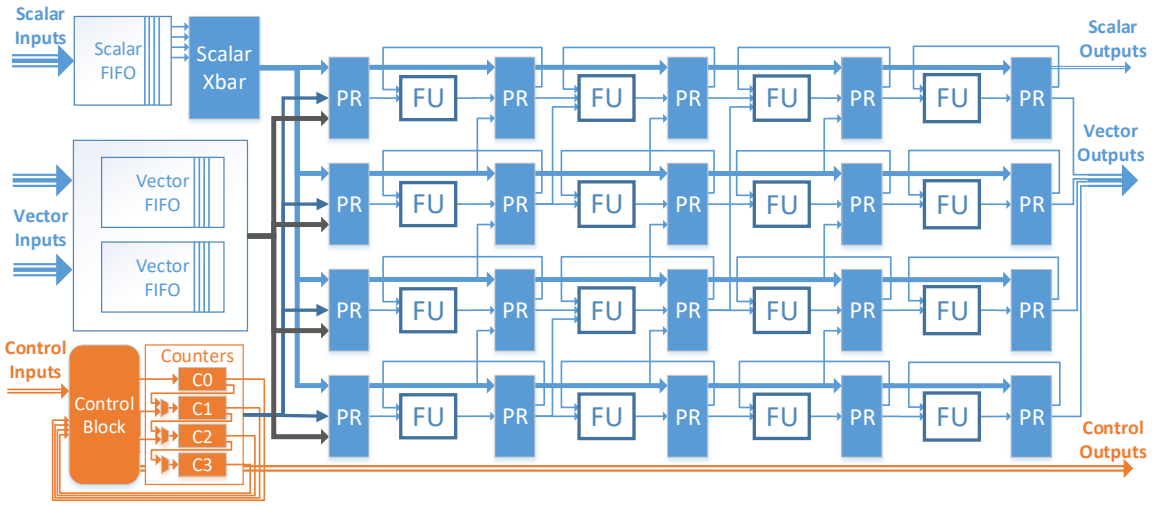


Figure 4.1: Pattern Compute Unit (PCU) architecture. We show only 4 stages and 4 SIMD lanes, and omit some control signals.

4.2.1 Pattern Compute Unit

The PCU is designed to execute a single, innermost parallel pattern in an application. As shown in Figure 4.1, the PCU datapath is organized as a multi-stage, reconfigurable SIMD pipeline. This design enables each PCU to achieve high compute density, and exploit both loop-level parallelism across lanes and pipeline parallelism across stages.

Each stage of each SIMD lane is composed of a *functional unit* (FU) and associated pipeline registers (PR). FUs perform 32 bit word-level arithmetic and binary operations, including support for floating point and integer operations. As the FUs in a single pipeline stage operate in SIMD, each stage requires only a single configuration register. Results from each FU are written to its associated register. PRs in each lane are chained together across pipeline stages to allow live values propagate between stages within the same lane. Cross-lane communication between FUs is captured using two types of intra-PCU networks: a reduction tree network that allows reducing values from multiple lanes into a single scalar, and a shift network which allows using PRs as sliding windows across stages to exploit reuse in stencil applications. Both networks use dedicated registers within PRs to minimize hardware overhead.

PCUs interface with the global interconnect using three kinds of inputs and outputs

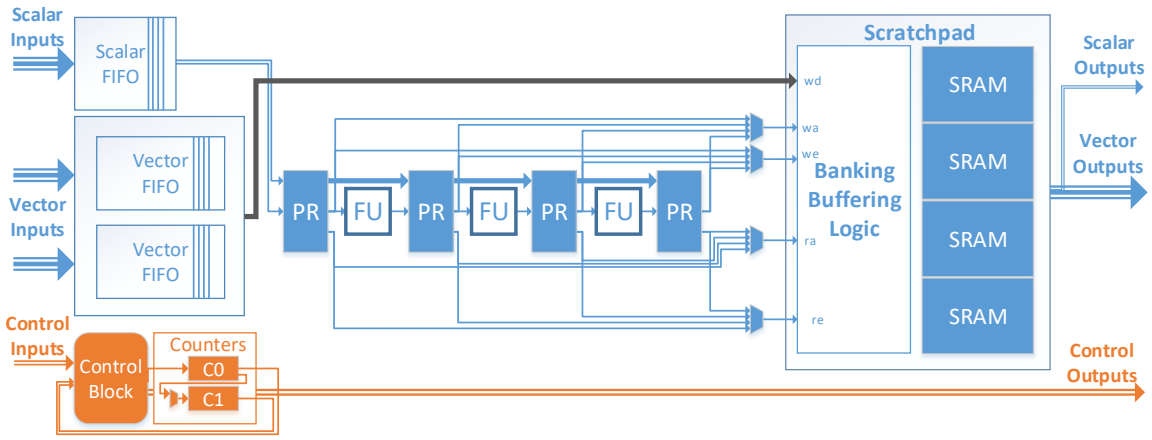


Figure 4.2: Pattern Memory Unit (PMU) architecture: configurable scratchpad, address calculation datapath, and control.

(IO); scalar, vector, and control. Scalar IO is used to communicate single words of data, such as the results of Folds. Each vector IO allows communicating one word per lane in the PCU, and is used in cases such as reading and writing to scratchpads in PMUs and transmitting intermediate data across a long pipeline between multiple PCUs. Each vector and scalar input is buffered using a small FIFO. Using input FIFOs decouples data producers and consumers, and simplifies inter-PCU control logic by making it robust to input delay mismatches. Control IO is used to communicate control signals such as the start or end of execution of a PCU, or to indicate backpressure.

A reconfigurable chain of counters generates pattern iteration indices and control signals to coordinate execution. PCU execution begins when the control block enables one of the counters. Based on the application's control and data dependencies, the control block can be configured to combine multiple control signals from both local FIFOs and global control inputs to trigger PCU execution. The control block is implemented using reconfigurable combinational logic and programmable up-down counters for state machines.

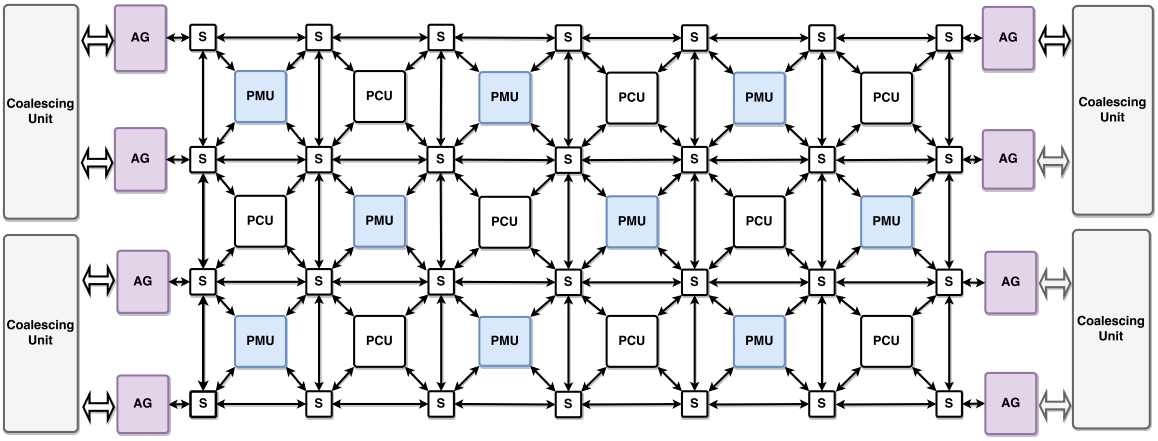


Figure 4.3: Plasticine chip-level architecture (actual organization 16 x 8). All three networks have the same structure.
PCU:Pattern Compute Unit, PMU: Pattern Memory Unit, AG: Address Generator, S: Switch Box.

4.2.2 Pattern Memory Unit

Figure 4.2 shows the architecture of a PMU. Each PMU contains a programmer-managed scratchpad memory coupled with a reconfigurable scalar datapath intended for address calculation. As shown in Figure 4.3, PMUs are used to distribute on-chip memory throughout the Plasticine architecture. Plasticine makes a distinction between the operations involved in memory addresses calculation and the core computation underlying applications. Address calculation is performed on the PMU datapath, while the core computation is performed within the PCU. Several observations have motivated this design choice: (i) Address calculation involves simple scalar math, which requires simpler ALUs than the FUs in PCUs; (ii) Using multiple lanes for address computation is often unnecessary for most on-chip access patterns; and (iii) Performing address calculation within the PCU requires routing the addresses from the PCU to the PMU, which occupies PCU stages and output links, and can lead to PCU under-utilization.

The scratchpads are built with multiple SRAM banks matching the number of PCU lanes. Address decoding logic around the scratchpad can be configured to operate in several banking modes to support various access patterns. *Strided banking* mode supports linear access patterns often found on dense data structures. *FIFO* mode supports streaming

accesses. *Line buffer* mode captures access patterns resembling a sliding window. *Duplication* mode, where the contents are duplicated across all memory banks, provides multiple read address channels to support parallelized on-chip gather operations.

Just as banking is important to feed multiple SIMD units to sustain compute throughput, *N-buffering*, or generalized double buffering, is just as important to support coarse-grained pipelines. The PMU scratchpad can be configured to operate as an N-buffer with any of the banking modes described. N-buffers are implemented by partitioning the address space in each SRAM bank into N disjoint regions. Using write and read state information, an appropriate offset is added to each bank's local address to access the correct data.

A programmable counter chain and control block triggers PMU execution similar to the PCU. Each PMU typically contains write address calculation logic from the producer pattern, and read address calculation logic from the consumer pattern. Based on the state of the local FIFOs and external control inputs, the control block can be configured to trigger the write address computation, read address computation, or both, by enabling the appropriate counters.

4.2.3 Interconnect

Plasticine supports communication between PMUs, PCUs, and peripheral elements using three kinds of interconnect - scalar, vector, and control. The networks differ in the granularity of data being transferred; scalar networks operate at word-level granularity, vector networks operate at multiple word-level granularity, and control networks operate at bit-level granularity. The topology of all three networks is identical, and is shown in Figure 4.3. All networks are statically configured. Links in network switches include registers to avoid long wire delays.

Applications commonly contain nested pipelines, where the outer pipeline levels only require counters and some reconfigurable control. In addition, as outer pipeline logic typically involves some level of control signal synchronization, they are control hotspots which require a large number of control and scalar inputs and outputs. To handle outer pipeline logic in an efficient manner, scalar and control switches share a reconfigurable control block and counters. Incorporating control logic within switches reduces routing to hotspots

and increases PCU utilization.

4.2.4 Off-chip Memory Access

Off-chip DRAM is accessed from Plasticine using 4 DDR memory channels. Each DRAM channel is accessed using several *address generators* (AG) on two sides of the chip, as shown in Figure 4.3. Each AG contains a reconfigurable scalar datapath to generate DRAM requests, similar in structure to the PMU datapath shown in Figure 4.2. In addition, each AG contains FIFOs to buffer outgoing commands, data, and incoming responses from DRAM. Multiple AGs connect to an address coalescing unit, which arbitrates between the AGs and processes memory requests.

AGs can generate memory commands that are either *dense* or *sparse*. Dense requests are used to bulk transfer contiguous DRAM regions, and are commonly used to read or write tiles of data. Dense requests are converted to multiple DRAM burst requests by the coalescing unit. Sparse requests enqueue a stream of addresses into the coalescing unit. The coalescing unit uses a coalescing cache to maintain metadata on issued DRAM requests and combines sparse addresses that belong to the same DRAM request to minimize the number of issued DRAM requests. In other words, sparse memory loads trigger a *gather* operation in the coalescing unit, and sparse memory stores trigger a *scatter* operation.

4.2.5 Control Flow

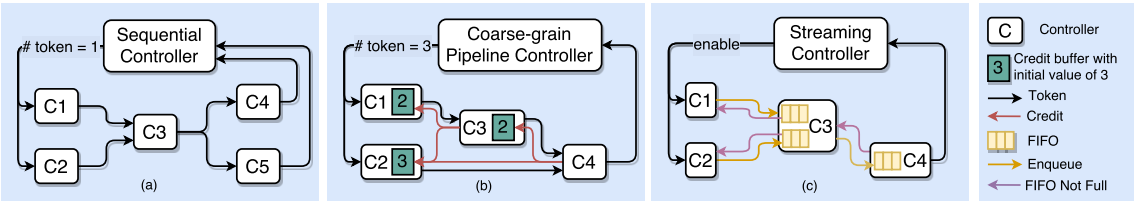


Figure 4.4: Sequential, coarse-grained pipelining, and streaming control schemes.

Plasticine uses a distributed and hierarchical control scheme that minimizes synchronization between units in order to adapt to limited bit-wise connectivity in the interconnect. Plasticine uses a distributed *token* and *credit* based protocol to orchestrate control flow between PCUs and PMUs. Tokens refer to feed-forward control flow signals, which are used

to track the amount of data a PCU or PMU currently has available to operate on. Credits are signals which indicate back-pressure. They tell the PCU how many elements it can produce before its consumers will no longer be able to buffer its outputs. A unit can therefore execute when it has at least one token from all of its inputs, and one credit from all consumer units that it is writing to.

We support three types of controller protocols inferred from our high-level language constructs: (a) *sequential* execution, (b) *coarse-grained pipelining*, and (c) *streaming* (Figure 4.4). These control schemes correspond to outer loops in the input program, and determine how the execution of individual units are scheduled relative to other units. Units are grouped into hierarchical sets of controllers. The control scheme of sibling controllers is based on the scheme of their immediate parent controller.

In a sequential parent controller, only one data dependent child is active at any time. This is commonly used when a program has loop-carried dependencies. To enforce data dependencies, we use *tokens*, which are feed-forward pulse signals routed through the control network. When the parent controller is enabled, a single token is sent to all *head* children with no data dependencies on their siblings. Upon completing execution, each child then passes its token to consumers of its output data. Each controller is enabled only when tokens from all dependent data sources are collected. Tokens from the last set of controllers, whose data are not consumed by any sibling controller in the same level of hierarchy, are sent back to the parent. The parent combines the tokens and either sends tokens back to the heads for the next iteration, or passes the token along at its own hierarchy level when all of its iterations have finished.

In coarse-grained pipelines, child controllers are executed in a pipelined fashion. To allow concurrent execution, the parent controller sends N *tokens* to the *heads*, where N is the number of data dependent children in the critical path. This allows all children to be active in the steady state. To allow producers and consumers to work on the same data across different iterations, each intermediate memory is M-buffered, where M is the distance between the corresponding producer and consumer on their data dependency path. To prevent producers from overflowing the down-stream buffer, each child controller handles backpressure by keeping track of available down-stream buffer sizes using *credits*. The number of credits is statically initialized to M. Each producer decrements its credit count

after producing all the data for the "current" iteration of the parent. Similarly, the consumer sends a credit back through the network after consuming all the data for the "current" iteration of the parent. In the coarse-grained pipelining scheme, each child is enabled when it has at least one token and one credit available.

Finally, child controllers with a *streaming* parent controller execute in a fine-grain pipelining fashion. This allows the compiler to fit a large inner pattern body by concatenating multiple units to form a large pipeline. In streaming mode, children communicate through FIFOs. A controller is enabled when all FIFOs it reads from are *not empty* and all FIFOs it writes to are *not full*. FIFOs are local to the consumer controller, so *enqueue* and *not empty* signals are routed from consumer to producer through the control network.

To enforce these control protocols, we implement specialized reconfigurable control blocks using statically programmable counters, state machines and combinational lookup tables. Each PCU, PMU, switch, and memory controller in the architecture has a control block. In general, controllers without any children are mapped to PCUs, while outer controllers are mapped to control logic in switches. This mapping gives outer controllers, which often have many children to synchronize with, a higher radix for communication. The hierarchy and distributed communication in Plasticine's control scheme allows the compiler to leverage the multiple levels of parallelism available in nested parallel patterns with only minimum overhead from bit-level reconfigurability.

4.2.6 Application Mapping

We begin with an application represented as a hierarchy of parallelizable dataflow pipelines written in DHDL [61]. Pipelines in DHDL are either *outer controllers* which contain only other pipelines, or *inner controllers* which contain no other controllers, only dataflow graphs of compute and memory operations.

To map DHDL to Plasticine, we first unroll outer pipelines using user-specified or auto-tuned parallelization factors. The resulting unrolled representation is then used to allocate and schedule *virtual* PMUs and PCUs. These virtual units are an abstracted representation of the units in Plasticine which have an infinite number of available inputs, outputs, registers, compute stages, and counter chains. As outer controllers contain no computation, only

control logic, they map to a virtual PCU with no compute stages, only control logic and counter chains. The computation in inner controllers is scheduled by linearizing the data flow graph and mapping the resulting list of operations to virtual stages and registers. Each local memory maps to a virtual PMU. Stages used to compute read and write addresses for this memory are copied to the virtual PMUs.

We then map each virtual unit into a set of physical units by partitioning its stages. Virtual PCUs are partitioned into multiple PCUs, while PMUs become one PMU with zero or more supporting PCUs. While graph partitioning is NP-hard in general, each virtual unit tends to have far less than 200 compute stages with very few cyclic dependencies. This means that a greedy algorithm with a few simple heuristics can reasonably approximate a perfect physical unit partitioning. In our partitioning algorithm, we use a cost metric which calculates the number of physical stages, live variables per stage, and scalar and vector input/output buses required for a given partitioning. Note that these communication and computation costs are always statically predictable because we begin with a full dataflow representation of the application. Using our heuristics, the compiler selects a proposed partitioning where all PCUs and PMUs are physically realizable given some chosen set of Plasticine architecture parameters (number of PCUs, PMUs, stages, lanes, buses, etc.) and which maximize the ALU and local memory utilization.

Following partitioning, we generate the control logic corresponding to the controller hierarchy as described in Section 4.2.5. We then perform hierarchical binding of virtual hardware nodes to physical hardware resources, including datapath and control path placement and routing, register allocation of SIMD units, including mapping stages to physical ALUs, and allocating scratchpads and control resources. The hierarchical nature of Plasticine allows us to dramatically reduce the search space with less than 1000 nodes in each level of mapping.

Given this placement and routing information, we then generate a Plasticine configuration description, akin to an assembly language, which is used to generate a static configuration "bitstream" for the architecture. The hierarchical architecture, coupled with the coarse granularity of buses between compute units, allows our entire compilation process to finish (or fail) in only a few minutes, as compared to the hours it can take to generate FPGA configurations.

	Component	Range	Final Value
Pattern Compute Unit	Lanes	4, 8, 16, 32	16
	Stages	1 – 16	6
	Registers/Stage	2 – 16	6
	Scalar Inputs	1 – 16	6
	Scalar Outputs	1 – 6	5
	Vector Inputs	1 – 10	3
	Vector Outputs	1 – 6	3
Pattern Memory Unit	Bank Size	4, 8, 16, 32, 64KB	16KB
	Scratchpad Banks	*Number of PCU Lanes*	16
	Total Scratchpad	*Bank size * banks*	256KB
	Stages	1 – 16	4
	Registers/Stage	2 – 16	6
	Scalar Inputs	1 – 16	4
	Scalar Outputs	0 – 6	0
	Vector Inputs	1 – 10	3
	Vector Outputs	1 – 6	1
Architecture	PCUs	—	64
	PMUs	—	64

Table 4.1: Design space and final selected parameters.

4.2.7 Architecture Sizing

Thus far, we have described a parameterized architecture composed of, among other things, PCUs, PMUs, and interconnect. We now describe our process for tuning the PCU and PMU parameters to create the final Plasticine architecture that we evaluate in Section 4.3.

Table 4.1 summarizes the architecture parameters under consideration, the possible values for each, and the final value we selected. To improve the probability of application routability, we restrict PMUs and PCUs to be homogeneous across the architecture.

In selecting design parameters, we first prune the space by analyzing the characteristics of the benchmarks listed in Table 4.2. Based on models of the performance of each

benchmark, we determine that the ideal inner controller parallelization factor across all benchmarks is between 8 and 32. In Plasticine, this corresponds to Pattern Compute Units with between 8 and 32 SIMD lanes. We select a balanced architecture with 16 lanes. Vectors of 16, 4 byte words also conveniently match our main memory's burst size of 64 bytes. For the PMU scratchpads, we found that ideal tile sizes for our benchmarks are at most 4000 words per bank. We therefore set the PMU to have 16 configurable, 16KB banks, for a total of 256KB per PMU.

We next search the remaining architectural space to select the number of stages, registers per stage, inputs, and outputs per PCU. In our programming model, parallelizing outer controllers corresponds in hardware to duplicating inner controllers. This means that we can assume that, to a first order, outer loop parallelization in a given application will not change its ideal PCU parameters, only the required number of PCUs. We therefore fix each benchmark with realistic parallelization factors and use these benchmarks to determine how to minimize the total PCU area while maximizing useful compute capacity. Note that we must also allow the number of required PCUs to vary, as these parameters directly impact how many physical PCUs a virtual PCU will require. Given the minimized PCU design, we can then create a Plasticine architecture with maximum performance for a given total chip area budget.

We use a model-driven, brute force search to tune each architectural parameter across different applications. To drive this search, we use benchmark-normalized *area overhead* as a cost metric for useful PCU area. When tuning a parameter, we sweep its value. For each proposed value, we sweep the remaining space to find the minimum possible PCU Area ($Area_{PCU}$). We then normalize these areas based on their minimum (Min_{PCU}) and report the overhead of each possible parameter value as $Area_{PCU}/Min_{PCU} - 1$. The area of a single PCU is modeled as the sum of the area of its control box, FUs, pipeline registers, input FIFOs, and output crossbars. The total number of PCUs required for a given set of design parameters is calculated using the mapping procedure outlined in Section 4.2.6.

In our studies, we found that the ordering of parameters during tuning made little difference to the final architectural design. For simplicity, we report a search procedure using one possible ordering, but any ordering would result in the same final parameter values.

We first examine the space defined by the number of stages per physical PCU. All

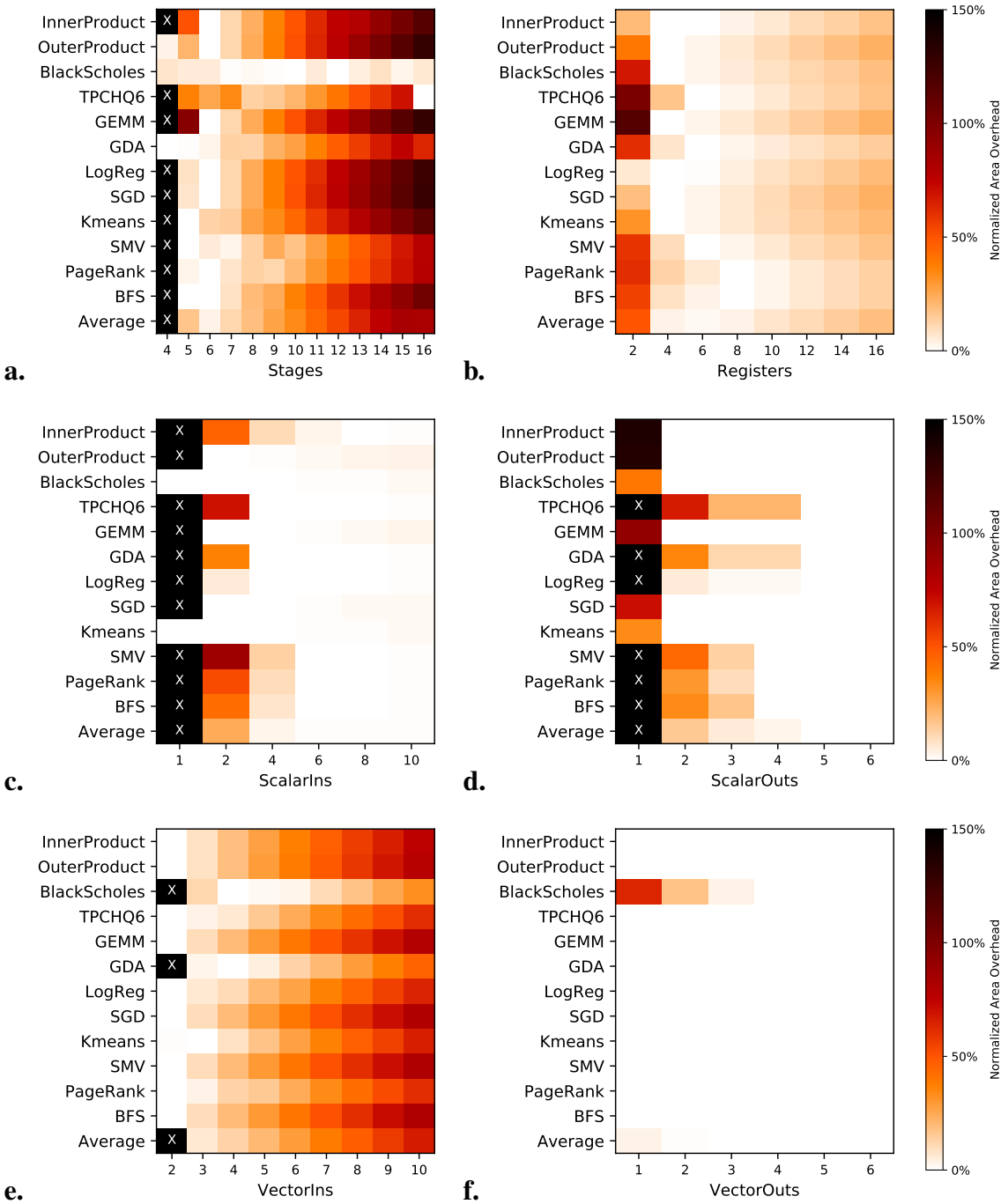


Figure 4.5: Area overhead ($Area_{PCU}/Min_{PCU} - 1$) while sweeping various Plasticine PCU parameters for a subset of our benchmarks. Min_{PCU} is benchmark-specific minimum possible area. Areas marked with an $\times$ denote invalid parameters for the given application. *a.* Stages per PCU; *b.* Registers per FU with 6 stages; *c.* Scalar inputs per PCU with 6 stages and 6 registers; *d.* Scalar outputs per PCU with 6 stages, 6 registers, and 6 scalar inputs; *e.* Vector inputs per PCU with 6 stages and 6 registers; and *f.* Vector outputs per PCU with 6 stages, 6 registers, and 3 vector inputs.

other parameters are left unrestricted. Figure 4.5a shows the estimated area overheads after sweeping the number of stages between 4 and 16. Here, we see that the ideal number of stages per PCU is 5 or 6 for most benchmarks. In these benchmarks, the amount of compute per pattern is fairly small, allowing patterns to be mapped to a single PCU. At least 5 stages are required for a full cross-lane reduction tree within the PCU. In BlackScholes, the core compute pipeline has around 80 stages. This is long enough that stages per PCU has little impact on average FU utilization. In TPCHQ6, the core computation is 16 stages long, meaning the area overhead is minimized at 8 and 16 stages (even divisors of the compute). We select 6 stages per PCU as a balanced architecture across all of our benchmarks. This choice means that applications like TPCHQ6 with a relatively small number of operations that does not divide evenly by 6 will underutilize PCU partitions, but this is an inevitable consequence of partitioning over homogeneous units.

We next determine the number of registers per FU. We again sweep the parameter space, fixing the number of stages at 6 but leaving all other parameters unrestricted. From Figure 4.5b, we see that the ideal number of registers across most applications is between 4 and 6. This directly corresponds to the maximum number of live values at any given point in each PCU's pipeline of stages. Below 4 registers, PCUs are constrained by the number of live values they can hold at a given time, causing extraneous partitioning. Above 8 registers per FU, the cost of the unused registers becomes noticeable relative to the total PCU area. We select 6 registers per FU.

Following the same procedure, we determine the number of scalar inputs and outputs. Scalar logic is relatively cheap, but, like registers, lack of available scalar inputs or outputs can cause logic to be split across many PCUs, creating large overheads from unutilized logic. Thus, we see in Figure 4.5(c,d) that each benchmark has some minimum number of inputs and outputs required, after which adding more of either has little impact. We select 6 scalar inputs and 5 scalar outputs, as this minimizes area overhead across all benchmarks.

Finally, we tune the vector inputs and outputs per PCU in the same manner. Vectors are tuned separately from scalars, as the two use different interconnect paths between PCUs and different registers within PCUs. Note here that vector inputs are associated with input FIFOs, which represent a sizeable fraction of PCU area. We therefore want to minimize vector inputs as much as possible. However, as seen in Figure 4.5e, due to limitations in

splitting across PCUs, BlackScholes and GDA are restricted to having at least 3 vector inputs. Figure 4.5f shows that vector outputs are relatively inexpensive and have little impact on required design area. We thus choose 3 vector inputs and 3 vector outputs per PCU.

Using a similar approach, we also select the PMU parameters given in Table 4.1. Note that the number of vector inputs and outputs for PMUs trivially correspond to the read, write, and data buses of the scratchpad. PMUs currently never use scalar outputs, as the compiler always maps the results of memory reads to vector buses.

After this tuning process, we now have a tuned PCU and PMU design. Based on profiling of each benchmark, we choose 16 $\times$ 8 units. We also experimented with multiple ratios of PMUs to PCUs. We choose a 1:1 ratio of PMUs to PCUs. While larger ratios (e.g. 2:1 PMUs to PCUs) improved unit utilization on some benchmarks, these ratios were less energy efficient.

4.3 Evaluation

In this section, we evaluate the performance and power efficiency of Plasticine against a commercial Stratix V FPGA. We compare the runtime and power of the Plasticine architecture to efficient FPGA implementations for benchmarks taken from the machine learning, data analytics, and graph processing domains. FPGAs are widely available with mature toolchain support, which makes it possible to obtain performance data on real hardware.

4.3.1 Benchmarks

We developed a set of benchmarks that stress a variety of properties of the two architectures, such as dense data processing and data-dependent memory access. We use real-world applications to guide the design of these benchmarks to ensure that Plasticine is capable of doing useful work. Table 4.2 provides a summary of the applications.

Among the dense applications, *inner product*, *outer product*, and *GEMM* (single precision general matrix multiplication) are fundamental linear algebra operations and at the core of many algorithms. *TPC-H Query 6* is a simple filter-reduce kernel that demonstrates

Benchmark	**Data Size(s)**	**Data Type**
Inner Product	768,000,000	float32
Outer Product	76,800 $\times$ 76,800	float32
Black-Scholes	96,000,000 entries	float32
TPC-H Query 6	960,000,000 entries	int32
GEMM	47 $\times$ 7,680 * 7,680 $\times$ 3,840	float32
GDA	3,840,000 points; 96 dims	float32
LogReg	5 iters; 1,536 points; 384 dims	float32
SGD	30 iters; 38,400 points; 768 dims	float32
Kmeans	50 iters; 1,536 points; 96 dims; K = 20	float32
CNN	model size 884736, data size 57600	float32
SMDV	3,840 $\times$ 3,840 with $E[NNZ]_{node} = 60$	float32
PageRank	100 iters; 7,680 pages	int32
BFS	$E[edges]_{node} = 8 \times$ 10 layers	int32

Table 4.2: Evaluation benchmarks.

database query functionality. *Black-Scholes* is a computation-heavy finance algorithm with extremely deep pipelines. Gaussian Discriminant Analysis (*GDA*) and Stochastic Gradient Descent (*SGD*) are common machine learning algorithms that involve relatively complicated memory accesses and exhibit many choices for parallelization. *K-means* clustering groups a set of input points by iteratively calculating the k best cluster centroids. K-means is written using a dense HashReduce to calculate the next iteration's centroids. Convolutional Neural Network (*CNN*) is an important machine learning kernel used for image classification. CNNs involve multiple layers of computation, where each layer involves several 3D convolution operations on an input feature map.

The sparse applications involve data-dependent accesses to memory and non-deterministic computation. Sparse matrix-dense vector (*SMDV*) multiplication is another fundamental linear algebra kernel used in many sparse iterative methods and optimization algorithms. *PageRank* is a popular graph algorithm that involves off-chip sparse data gathering to iteratively update page rankings. Breadth-First Search (*BFS*) is another graph algorithm that performs a data-dependent, frontier-based traversal and uses data scatters to store information about each node.

We implement each of these benchmarks in DHDL, which was described in chapter 3.

4.3.2 Plasticine Design

We implement the Plasticine architecture in Chisel [68] using the selected parameters listed in Table 4.1. The architecture is organized as a 16×8 array of units, with a 1:1 ratio of PMUs to PCUs. This design is synthesized using Synopsys Design Compiler with a 28nm technology library. Critical paths in the design have been optimized for a clock frequency of 1 GHz. Total chip area estimates are obtained after synthesis. Local scratchpad and FIFO sizes are obtained using Synopsys Memory Compiler with a 28nm library. We profile single PCU, PMU, and AG power using Synopsys PrimeTime with RTL traces. Static power for the entire chip and dynamic power for utilized units are included in the total power. Table 4.3 provides the component-wise area breakdown for Plasticine at an area footprint of 112.77mm^2. The final Plasticine architecture has a peak floating point performance of 12.3 single-precision TFLOPS and a total on-chip scratchpad capacity of 16 MB.

Execution times for Plasticine are obtained from cycle-accurate simulations performed using Synopsys VCS coupled with DRAMSim2 [69] to measure off-chip memory access times. We configure DRAMSim2 to model a memory system with 4 DDR3-1600 channels, giving a theoretical peak bandwidth of 51.2 GB/s.

We modify the DHDL compiler to generate static configurations for Plasticine using the procedure outlined in Section 4.2.6. Using the modified compiler, each benchmark is compiled to a Plasticine configuration, which is used to program the simulator. Total reported runtime for Plasticine begins after data is copied into the accelerator's main memory, and ends when execution on the accelerator is complete (i.e. before copying data back to the host).

4.3.3 Plasticine Design Overheads

We first examine the area overheads of design decisions within the Plasticine architecture. Each decision is isolated and evaluated based on an idealized architecture. These architectures are normalized such that, given a 1 GHz clock and fixed local memory sizes, the performance of each benchmark is the same as its performance on the final Plasticine architecture. These design decisions (columns *a.* — *e.* in Table 4.4) allow an arbitrary number of PCUs and PMUs for each benchmark. This is done to isolate the quality of the choice from an application's utilization of a fixed size architecture.

We first evaluate the cost of partitioning an application into coarse-grain PCUs and PMUs. Here, we compare the projected areas of benchmark-specific ASIC designs to an idealistic Plasticine architecture with *heterogeneous* PCUs and PMUs. For a given benchmark, ASIC area is estimated as the sum of the area of its compute and memory resources. The area of each of these resources was characterized using Synopsys DC. The Plasticine architecture has all of the features described in Section 4.2, including configuration logic, configurable banked memories, and statically configurable ALUs. Column *a.* of Table 4.4 shows the projected costs of such a heterogeneous architecture relative to the projected area of the benchmark-specific chip design. Relative to ASIC designs, the area overhead of reconfigurable units is on average about 2.8×. This is the base overhead of Plasticine's reconfigurability, primarily concentrated in making memory controllers configurable and

	Component	Area (mm^2)	Area $(\%)$
PCU (48.16%)	FUs	0.622	73.32
	Registers	0.144	16.97
	FIFOs	0.082	9.65
	Control	0.001	0.06
	Total (single PCU)	0.849	100.00
PMU (30.2%)	Scratchpad (256KB)	0.477	89.73
	FIFOs	0.024	4.52
	Registers	0.023	4.28
	FUs	0.007	1.29
	Control	0.001	0.18
	Total (single PMU)	0.532	100.00
Interconnect (16.66%)		18.796	100.00
Memory Controller (4.98%)	4 Coalescing Units, 34 AGs	5.616	100.00
Plasticine	64 PCUs, 64 PMUs, Memory Controller, Interconnect	112.796	100.00

Table 4.3: Plasticine area breakdown.

Benchmark	Coarse *a.*	Homogeneous *b.* PMUs		Homogeneous *c.* PCUs		Generalized *d.* PMUs		Generalized *e.* PCUs	
Inner Product	*2.64*	*1.21*	(3.18)	*2.66*	(8.45)	*1.53*	(12.92)	*1.02*	(13.18)
Outer Product	*1.54*	*2.07*	(3.18)	*1.83*	(5.81)	*1.00*	(5.81)	*1.02*	(5.95)
Black-Scholes	*2.05*	*1.05*	(2.15)	*1.59*	(3.43)	*1.18*	(4.04)	*1.10*	(4.46)
TPCH-Query 6	*2.26*	*1.15*	(2.59)	*3.90*	(10.10)	*1.24*	(12.49)	*1.15*	(14.32)
GEMM	*1.63*	*1.45*	(2.36)	*1.62*	(3.82)	*1.00*	(3.83)	*1.02*	(3.92)
GDA	*1.95*	*1.79*	(3.50)	*3.03*	(10.59)	*1.34*	(14.19)	*1.01*	(14.38)
LogReg	*1.55*	*1.91*	(2.96)	*1.73*	(5.12)	*1.00*	(5.13)	*1.02*	(5.20)
SGD	*7.67*	*1.09*	(8.40)	*1.82*	(15.31)	*1.41*	(21.61)	*1.02*	(21.98)
Kmeans	*2.81*	*1.88*	(5.29)	*1.74*	(9.19)	*1.00*	(9.20)	*1.02*	(9.42)
SMDV	*5.03*	*1.24*	(6.26)	*4.04*	(25.31)	*1.36*	(34.51)	*1.06*	(36.73)
PageRank	*7.14*	*1.18*	(8.41)	*3.39*	(28.51)	*1.46*	(41.73)	*1.03*	(42.83)
BFS	*2.91*	*1.38*	(4.02)	*2.14*	(8.61)	*1.21*	(10.40)	*1.03*	(10.70)
GeoMean	*2.77*	*1.41*	(3.92)	*2.32*	(9.07)	*1.21*	(11.00)	*1.04*	(11.46)

Table 4.4: Estimated *successive* and (cumulative) area overheads of *a.* generalizing ASICs into reconfigurable, heterogeneous PCUs and PMUs; *b.* restricting the architecture to homogeneous PMUs; *c.* further restricting the architecture to homogeneous PCUs; *d.* generalizing homogeneous PMUs across applications; *e.* generalizing homogeneous PCUs across applications.

converting compute logic from fixed operations to reconfigurable FUs.

While use of heterogeneous units is ideal for area utilization, it creates an intractable mapping problem and does not generalize across different applications. In column *b.* of Table 4.4, we show the cost of moving from a heterogeneous architecture to an architecture still with heterogeneous PCUs but a single homogeneous PMU design. We still allow this PMU design to be unique for each benchmark, but within a single benchmark we size the PMU scratchpads based on the largest scratchpad the program requires. The average overhead from moving to uniform PMUs is 1.4×, with particularly large overheads for applications with drastically varying memory sizes. OuterProduct, for example, uses local memories for tiles of vectors of size N and an output tile of size N^2. In ASIC design, static knowledge about the target application allows specialization of each local memory to exactly the size and number of banks needed, thus saving a significant amount of area on SRAM. In Plasticine, we opt for uniformly sized memory units as they simplify mapping

and improve the likelihood that a given application will be routable.

Column *c.* shows the overheads of further restricting the PCUs to also be homogeneous, but still vary across benchmarks. The overhead here is particularly high for applications like PageRank with a large number of sequential loops. The bodies of all patterns are mapped to PCUs, but because each PCU is fixed to 16 lanes, most of the lanes in sequential loops, and therefore most of the area, is unused, leading to overheads of up to 8.4×. Similarly, applications like TPCHQ6 with widely varying compute pipeline lengths tend to underutilize the stages within homogeneous PCUs.

We next show the area overheads after selecting a single set of PMU parameters across all applications. As described in Section 4.2.7, this sets the total size of scratchpads in all benchmarks to 256KB each. While this local memory capacity is essential to the performance of applications like GEMM and OuterProduct [7], other applications have much smaller local memory requirements. As seen in column *d*, this unutilized SRAM capacity has an average chip area overhead of 1.2×.

Column *e.* lists the results of also generalizing PCUs across applications using the values obtained in Section 4.2.7. Here, we see that the remaining overhead is small compared to the cost of homogenizing the units, with an average of only 5% and a maximum of 15% for BlackScholes. This suggests that much of the variation in PCU requirements across applications is already represented by the variety of loops within a single application. This in turn makes generalization of compute across applications relatively inexpensive.

Cumulatively, we estimate that our homogeneous, generalized, unit based architecture has an average area overhead of 3.9× to 42.8× compared to application-specific chip designs with the same performance. This overhead of course varies significantly based on the local memory and compute requirements of the benchmark. While the PCU and PMU utilizations of the final, fixed size Plasticine architecture, later shown in Table 4.6, tend to be less than 100%, we do not view this in itself as an area overhead. Instead, the Plasticine architecture is considered a "sufficiently large" fabric which can be used to implement the ideal architectures listed in column *e.*, and the remaining unit resources can be clock gated.

Benchmark	Power (W)		Plasticine / FPGA		
	FPGA	Plasticine	Power	Performance	Perf/W
Inner Product	21.8	18.9	0.9	1.4	1.6
Outer Product	24.4	26.9	1.1	6.7	6.1
Black-Scholes	28.3	24.7	0.9	5.1	5.8
TPCH-Query 6	21.7	20.5	0.9	1.4	1.5
GEMM	25.6	34.6	1.4	33.0	24.4
GDA	26.5	41.0	1.5	40.0	25.9
LogReg	22.9	28.6	1.2	11.4	9.2
SGD	25.6	10.7	0.4	6.7	15.9
Kmeans	23.9	12.9	0.5	6.1	11.3
CNN	34.4	42.6	1.2	95.1	76.9
SMDV	21.5	19.3	0.9	8.3	9.3
PageRank	21.9	17.1	0.8	14.2	18.2
BFS	21.9	14.0	0.6	7.3	11.4

Table 4.5: Power, performance, and performance-per-Watt comparisons between Plasticine and FPGA.

4.3.4 FPGA Design

We next compare the performance and power of Plasticine to an FPGA. We use the DHDL compiler to generate VHDL, which in turn is used to generate a bitstream for the FPGA using Altera's synthesis tools. We run each synthesized benchmark on an Altera 28nm Stratix V FPGA, which interfaces with a host CPU controller via PCIe. The FPGA has a 150 MHz fabric clock, a 400 MHz memory controller clock, and 48 GB of dedicated off-chip DDR3-800 DRAM with 6 channels and a peak bandwidth of 37.5 GB/s. Execution times for the FPGA are reported as an average of 20 runs. Like Plasticine, timing starts after copy of data from the host to the FPGA's dedicated DRAM completes and finishes when FPGA execution completes. We also obtain FPGA power estimates for each benchmark using Altera's PowerPlay tool after benchmark placement and routing.

	Utilization (%)						
	FPGA		Plasticine				
Benchmark	**Logic**	**Memory**	**PCU**	**PMU**	**AG**	**FU**	**Register**
Inner Product	24.3	33.5	17.2	25.0	47.1	69.8	10.2
Outer Product	38.2	71.4	15.6	46.9	88.2	21.6	12.8
Black-Scholes	68.9	100.0	65.6	21.9	41.2	25.1	53.4
TPCH-Query 6	24.3	33.4	28.1	25.0	47.1	70.8	20.2
GEMM	40.4	94.8	34.4	68.8	97.1	56.0	8.6
GDA	53.6	96.8	89.1	87.5	44.1	8.1	11.2
LogReg	28.4	73.4	51.6	68.8	8.8	30.2	12.3
SGD	60.1	58.2	6.3	9.4	8.8	34.3	7.2
Kmeans	42.1	65.4	10.9	17.2	8.8	35.5	10.9
CNN	86.8	99.0	48.9	98.4	100.0	62.5	25.0
SMDV	27.3	31.0	43.8	15.6	29.4	10.4	2.7
PageRank	31.3	33.4	28.1	20.3	20.6	3.9	1.9
BFS	25.3	45.9	18.8	15.6	11.8	3.1	1.5

Table 4.6: Resource utilization comparisons between Plasticine and FPGA.

4.3.5 Plasticine versus FPGA

Table 4.5 shows the power, performance, and performance-per-Watt of Plasticine relative to the Stratix V FPGA across our set of benchmarks. The table shows that Plasticine achieves higher energy efficiency over an FPGA. Table 4.6 shows the resource utilization on both platforms for each benchmark. We discuss individual benchmark results below.

Inner product and TPC-H Query 6 both achieve speedups of 1.4×, respectively. Both benchmarks are memory bandwidth bound, where a large amount of data is streamed from DRAM through a datapath with minimal compute. Hence, the performance difference corresponds to the difference in the achievable main memory bandwidth on the respective platforms. The power consumption on Plasticine is comparable to the FPGA as well, as a majority of PCUs and half the PMUs are unused and therefore power gated.

Outer product is also bandwidth bound, but contains some temporal locality, and hence can benefit from larger tile sizes. The FPGA is limited by the number of large memories

with many ports that it can instantiate, which in turn limits exploitable inner loop parallelism and potential overlap between compute and DRAM communication. Native support for banked, buffered scratchpads allows Plasticine to better exploit SIMD and pipelined parallelism, thereby achieving a speedup of 6.7×.

Black-Scholes streams several floating point arrays from DRAM through a pipeline of floating point operations. The large amount of floating point operations per DRAM access makes it compute-bound on most architectures. While the deeply pipelined nature of Black-Scholes makes it an ideal candidate for FPGA acceleration, the FPGA runs out of area to instantiate compute resources long before it can saturate its main memory bandwidth. Plasticine, on the other hand, has a much higher floating point unit capacity. Black-Scholes on Plasticine can be sufficiently parallelized to the point of being memory bound. From Table 4.6, we can see that using 65% of PCUs, Black-Scholes maximizes DRAM bandwidth utilization, achieving a speedup of 5.1×.

GEMM and GDA are compute-bound, with ample temporal and spatial locality. On Plasticine, GEMM achieves a speedup of 33.0×. GDA performs similarly with a speedup of 40.0×. Plasticine can exploit greater locality by loading larger tiles into the banked scratchpads, and hides DRAM communication latency by configuring scratchpads as double buffers. On the FPGA, creating banked, double-buffered tiles exhausts BRAM resources before compute resources, thereby limiting compute throughput. In the current mapping of GEMM to Plasticine, each PCU multiplies two tiles by successively performing pipelined inner products in its datapath. Parallelism is achieved within the PCU across the lanes, and across PCUs where multiple tiles are processed in parallel. More parallelism is achieved by processing more input tiles in parallel. As a result, in the current scheme GEMM performance is limited by the number of AGs, as more AGs are required to load multiple input tiles. In addition, since each PCU performs an inner product, FUs that are not part of the reduction network are under-utilized. More sophisticated mapping schemes, along with more hardware support for inter-stage FU communication within PCUs, can further increase compute utilization and hence improve performance [70].

CNN is another compute-intensive benchmark where Plasticine outperforms the FPGA, in this case by 95.1×. Plasticine's performace is due to much higher compute density and its ability to capture the locality of kernel weights and partial results within PMUs. To

efficiently exploit sliding window reuse in CNN, scratchpads are configured as line buffers to avoid unnecessary DRAM refetches. Each PCU performs a single 3D convolution by reading the kernel weights from a PMU and producing the output feature map into another PMU. The shift network between FUs in the PCUs enables data reuse within sliding windows and accumulation of partial sums within the pipeline registers, which minimizes scratchpad reads and writes. CNN is currently mapped onto Plasticine such that each PCU requires 2 PMUs; one PMU to hold kernel weights, the other PMU to store output feature map. As Plasticine is configured with 1:1 PCU:PMU ratio, this caps the PCU utilization at 49.0% while maximizing PMU and AG utilization. More optimized mapping using greater PMU sharing could overcome this limitation.

LogReg is a compute heavy benchmark where large tile sizes are used to capture locality. Parallelism is exploited at the outer loop level by processing multiple tiles in parallel, and at the inner loop using SIMD lanes within PCUs. Currently, the compiler exploits inner loop parallelism only within the SIMD lanes of a single PCU, and does not split the inner loop across multiple PCUs. Plasticine achieves a speedup of 11.4× by processing more input tiles in parallel at a faster clock rate than the FPGA.

SGD and Kmeans have sequential outer loops and parallelizable inner loops. The inherently sequential nature of these applications results in a speedup of 6.7× and 6.1× respectively on Plasticine, largely due to Plasticine's higher clock frequency. However, as Plasticine needs only a few PCUs to exploit the limited parallelism, most of the unused resources can be power gated, resulting in performance-per-Watt improvements of 39.8× and 12.3× respectively.

SMDV, PageRank, and BFS achieve speedups of 8.3×, 14.2×, and 7.3× respectively on Plasticine. Performance of these sparse benchmarks is limited by DDR random access DRAM bandwidth. SMDV and PageRank perform only sparse loads (gather), while BFS performs a gather and a scatter in every iteration. Scatter and gather engines are implemented on the FPGA using soft logic. The outer loop of these benchmarks is parallelized to generate multiple parallel streams of sparse memory requests, which maximizes the number of outstanding memory requests after address coalescing. The FPGA platform used in the baseline is limited in its random access DRAM bandwidth, as all the channels operate in 'ganged' mode as one wide DRAM channel. FPGA platforms with

multiple independent DRAM channels can in theory perform better than our FPGA baseline for sparse applications. However, scatter-gather units still have to be implemented in soft logic. Scatter-gather units require large amounts of local memory, but local memories (BRAM) are often a critical resource on FPGAs that can limit the number of outstanding memory requests and the efficacy of address coalescing. In addition, FPGA fabric clocks are typically slower than DRAM clocks, creating another bottleneck in harnessing random access bandwidth. Dedicated hardware like the coalescing units in Plasticine allows DRAM bandwidth to be used in a much more efficient manner.

In summary, Plasticine can maximize DRAM bandwidth utilization for streaming applications like Inner Product and TPC-H Q6, and sustain compute throughput for deeply pipelined datapaths to make applications like Black-Scholes memory-bound. Plasticine captures data locality and communication patterns in PMUs and inter-PCU networks for applications like GEMM, GDA, CNN, LogReg, SGD, and Kmeans. Finally, by supporting a large number of outstanding memory requests with address coalescing, DRAM bandwidth is effectively utilized for scatter and gather operations in SMDV, Pagerank, and BFS.

4.4 Conclusion

In this chapter we addressed the issue of FPGA architectural inefficiencies by introducing a new architecture called Plasticine. Plasticine is a novel reconfigurable architecture for efficiently executing both sparse and dense applications composed of parallel patterns. We identify the key computational patterns needed to capture sparse and dense algorithms and describe coarse-grained Pattern and Memory Compute Units capable of executing parallel patterns in a pipelined, vectorized fashion. These units exploit information about hierarchical parallelism, locality and memory access patterns within our programming model. We then use design-space exploration to guide the design of the Plasticine architecture and create a full software-hardware programming stack to map applications to an intermediate representation, which is then executed on Plasticine. We show that, for an area budget of 113 mm^2, Plasticine provides up to 95$\times$ improvement in performance and up to 77$\times$ improvement in performance-per-Watt compared to an FPGA in a similar process technology.

Chapter 5

Related Work

Several decades of excellent research in many related areas has produced a rich body of relevant work. This chapter discusses related work under several categories in relation to the work described in this dissertation.

5.1 Optimizing Compilers

Several research contributions have enriched the field of optimizing compiler transformations, both to conventional architectures such as CPUs as well as accelerators such as FPGAs. We classify work in this area under two categories; *tiling*, and *hardware from high-level languages*.

5.1.1 Tiling

Previous work on automated loop tiling has largely focused on tiling imperative programs using polyhedral analysis [71, 72]. There are many existing tools—such as Pluto [73], PoCC [74], CHiLL [75], and Polly [76]—that use polyhedral analysis to automatically tile and parallelize programs. These tools restrict memory accesses within loops to affine functions of the loop iterators. As a consequence, while they perform well on affine sections of programs, they fail on commonly occurring data-dependent operations like *filters* and

groupBys [77]. In order to handle these operations, recent work has proposed using preprocessing steps which segment programs into affine and non-affine sections prior to running polyhedral analysis tools [78].

While the above work focused on the analysis of imperative programs, this dissertation analyzes functional parallel patterns, which offer a strictly higher-level representation than simple imperative *for* loops. The additional semantic information available in patterns like *groupBy* and *filter*, enables parallel patterns to be automatically tiled using simple transformation rules, without the restriction that all memory accesses are purely affine. Little previous work has been done on automated tiling of functional programs composed of arbitrarily nested parallel patterns. Hielscher proposes a set of formal rules for tiling parallel operators *map*, *reduce*, and *scan* in the Parakeet JIT compiler, but these rules can be applied only for a small subset of nesting combinations [79]. Spartan [50] is a runtime system with a set of high-level operators (e.g., *map* and *reduce*) on multi-dimensional arrays, which automatically tiles and distributes the arrays in a way that minimizes the communication cost between nodes in cluster environments. In contrast to our work, Spartan operates on a tiled representation for distributed CPU computation and does attempt to optimize performance on individual compute units.

5.1.2 Hardware From High-Level Languages

Generating hardware from high-level languages has been widely studied for decades. CHiMPS [80] generates hardware from ANSI C code by mapping each C language construct in a dataflow graph to an HDL block. Kiwi [81] translates a set of C# parallel constructs (e.g., *event*, *monitor*, and *lock*) to corresponding hardware units. Bluespec [82] generates hardware from purely functional descriptions based on Haskell. Chisel [68] is an embedded language in Scala for hardware generation. AutoPilot [83] is a commercial HLS tool that generates hardware from C/C++/SystemC languages. Despite their success in raising the level of abstraction compared to hardware description languages, programmers are still required to write programs at a low-level and express how computations are pipelined and parallelized. In contrast, the work in this dissertation raises the level of abstraction, and applies compiler transformations to automatically pipeline and parallelize operations and

exploit on-chip memory for locality.

Existing hardware synthesis tools are limited in their ability to automatically infer and generate coarse-grained pipelines. A traditional software pipelining approach is typically used on innermost loop bodies consisting only of primitive operations. Optimizations like *unroll-and-jam*, and *unroll-and-squash* [84] also attempt to exploit pipelined parallelism, but target outer parallel loops with inner sequential loops. To pipeline imperfectly nested loops, some commercial high-level synthesis tools like Vivado [56] unroll all inner loops and then employ traditional software pipelining. Not only does this approach generate needlessly large designs for large benchmarks, it also suffers from long compilation times due to the complexity in scheduling a large number of unrolled instructions.

Several methods have been proposed to work around such limitations. Most of these methods are centered around generating the required HLS code that would have otherwise been hard to write manually, from a high-level language. For example, one study produces HLS C code from parallel patterns to target FPGAs [26]. By exploiting the access patterns of nested patterns to store sequential memory accesses to on-chip memory and parallelizing the computation with strip-mining, the compiler can generate hardware that efficiently utilizes memory bandwidth. A more recent study [85] produces optimized HLS code from Halide programs. This study proposes interesting optimizations such as *loop perfection* to improve pipelining efficiency, where imperfectly nested loops are converted to perfectly nested loops by moving computations into inner loop scopes. Other works like ElasticFlow [86] and CGPA [87] generate coarse-grained pipelines using FIFOs in between stages for communication. However, they handle only a restricted form of data access patterns and a restricted number of nesting levels. Our metapipelining technique is more general than previous approaches because: (i) metapipeline stages are decoupled using double buffers, therefore not restricting data access patterns, (ii) metapipelines are easily composed and nested to any number of levels, and (iii) metapipelines can handle dynamic rate mismatches as they use asynchronous handshaking for inter-stage synchronization, thereby obviating the need to calculate static initiation interval as well as knowing loop trip counts ahead of time.

Recent work has explored using polyhedral analysis with HLS to optimize for data

locality on FPGAs [88]. Using polyhedral analysis, the compiler is able to promote memory references to on-chip memory and parallelize independent loop iterations with more hardware units. However, the compiler is not able to analyze loops that include non-affine accesses, limiting the coverage of applications that can be generated for hardware. Our work can handle parallel patterns with non-affine accesses by inferring required hardware blocks (e.g., FIFOs, scatter-gather units) for non-affine accesses, while aggressively using on-chip memory for affine parts.

As high-level parallel patterns become increasingly popular to overcome the shortcomings of C based languages, researchers have recently studied generating hardware from functional parallel patterns. Lime [51] embeds high-level computational patterns (e.g., *map*, *reduce*, *split*, and *join*) in Java and automatically targets CPUs, GPUs, and FPGAs without modifying the code. Our compiler manages a broader set of parallel patterns (e.g., *groupBy*) and applies transformations even when patterns are nested, which is common in a large number of real-world applications.

5.2 Coarse-Grained Reconfigurable Architectures

As mentioned in Chapter 1, several researchers and industry practitioners alike have explored various flavors of coarse-grained building blocks to build reconfigurable architectures. Several surveys [33, 89, 90] provide a broad overview of prior work on CGRAs. We discuss a few relevant bodies of work under the following categories:

5.2.1 CGRAs With Reconfigurable Scratchpads

Several of the previously proposed reconfigurable fabrics lack support for reconfigurable, distributed scratchpad memories. Without the ability to reconfigure the on-chip memory system with the different banking and buffering strategies needed to support parallel patterns, the memory system becomes the bottleneck for many workloads.

For example, ADRES [34], DySER [36], Garp [91], and Tartan [38] closely couple a reconfigurable fabric with a CPU. These architectures access main memory through the

cache hierarchy shared with the host CPU. ADRES and DySER tightly integrate the reconfigurable fabric into the execution stage of the processor pipeline, and hence depend on the processor's load/store unit for memory accesses. ADRES consists of a network of functional units, reconfigurable elements with register files, and a shared multi-ported register file. DySER is a reconfigurable array with a statically configured interconnect designed to execute innermost loop bodies in a pipelined fashion. However, dataflow graphs with back-edges or feedback paths are not supported, which makes it challenging to execute patterns such as *Fold* and nested parallel patterns. Garp consists of a MIPS CPU core and an FPGA-like coprocessor. The bit-level static interconnect of the co-processor incurs the same reconfiguration overheads as a traditional FPGA, restricting compute density. Piperench [37] consists of a pipelined sequence of "stripes" of functional units (FUs). A word-level crossbar separates each stripe. Each FU has an associated register file which holds temporary results. Tartan [38] consists of a RISC core and an asynchronous, coarse-grained reconfigurable fabric (RF). The RF architecture is hierarchical with a dynamic interconnect at the topmost level, and a static interconnect in the inner level. The architecture of the innermost RF core is modeled after Piperench [37].

5.2.2 Architectures With Reconfigurable Datapaths

Architectures with reconfigurable functional units consume less power as they do not incur the overheads of traditional instruction pipelines such as instruction fetch, decode, and register file access. These overheads account for about 40% of the datapath energy on the CPU [11] and about 30% of the total dynamic power on the GPU [12]. Furthermore, using a reconfigurable datapath in place of a conventional instruction pipeline in a GPU reduces energy consumption by about 57% [13]. TRIPS [92] is a tiled architecture where execution proceeds dynamically in a dataflow fashion, while instructions are statically issued within a block. TRIPS does not have a static interconnect, but contains two dynamic interconnection networks [93]: an operand network (OPN) to route operands between tiles, and an on-chip network (OCN) to communicate with cache banks. The Raw microprocessor [94] is a tiled architecture where each tile consists of a single-issue in-order processor, a floating point unit, a data cache, and a software-managed instruction cache. Tiles communicate with their

nearest neighbors using pipelined, word-level static and dynamic networks. Plasticine does not incur the overheads of dynamic networks and general purpose processors mentioned above. Using hardware managed caches in place of reconfigurable scratchpads reduces power and area efficiency in favor of generality.

5.2.3 Dense Datapaths and Hierarchical Pipelines

Plasticine's hierarchical architecture, with dense pockets of pipelined SIMD functional units and decentralized control, enables capturing a substantial amount of data communication within PCUs and efficiently exploiting coarse-grained pipeline parallelism in applications. In contrast, architectures that lack hierarchal support for nested pipelining in the architecture use their global interconnect to communicate most results. Hence, the interconnect can be a bandwidth, power, or area bottleneck. For example, RaPiD [95] is a one-dimensional array of ALUs, registers, and memories with hardware support for static and dynamic control. A subsequent research project called Mosaic [96] includes a static hybrid interconnect along with hardware support to switch between multiple interconnect configurations. RaPiD's linear pipeline enforces a rigid control flow which makes it difficult to exploit nested parallelism. HRL [39] combines coarse-grained and fine-grained logic blocks with a hybrid static interconnect. While a centralized scratchpad enables some on-chip buffering, the architecture is primarily designed for memory-intensive applications with little locality and nested parallelism. Triggered instructions [40] is an architecture consisting of coarse-grained processing elements (PEs) of ALUs and registers in a static interconnect. Each PE contains a scheduler and a predicate register to implement dataflow execution using triggers and guarded actions. The control flow mechanism used in Plasticine has some similarities with Triggered instructions. While this architecture has the flexibility to exploit nested parallelism and locality, the lack of hierarchy increases communication over the global interconnect which can create bottlenecks, and reduces compute density in the datapath. Wavescalar [97] is another tiled dynamic dataflow architecture with four levels of hierarchy, connected by dynamic interconnects that vary in topology

and bandwidth at each level. While execution is dataflow driven, the datapath is not reconfigurable, and broadcast and dynamic interconnects are used for communication. Coarse-grained parallelism can be exploited using multi-threaded support and barriers to achieve synchronization. However, lack of a distributed scratchpad means that parallel memory accesses are serialized at the memory interface.

5.2.4 Statically scheduled interconnects

Some architectures allow interconect configurations to change periodically based on a statically determined schedule, to allow for greater interconnect link utilization compared to a fully static network [98–100]. Such interconnects typically require the compiler to provide a valid static schedule using modulo scheduling. While this approach is effective for inner loops with predicatable latencies and fixed Initiation Interval (II), variable latency operations and hierarchical loop nests complicates the compiler by creating scheduling complexities to arrive at a single module schedule. HyCube [41] has a similar statically scheduled network with the added ability to bypass intermediate switches in the same cycle. This approach allows operands to travel multiple hops in a single cycle, but creates long wires and combinational paths, which adversely affects the clock period and scalability. Long combinational paths through the interconnect are avoided in Plasticine by pipelining the interconnect switches.

Chapter 6

Conclusions

As technology scaling limitations and energy efficiency requirements force architects to pursue specialized alternatives, reconfigurable architectures like FPGAs hold promise to satisfy modern computing demands. However, fine-grained reconfigurability in FPGAs incurs programming and architectural inefficiencies which has hindered widespread adoption for many years.

This dissertation has described practical techniques to address the programming and architectural inefficiencies of FPGAs. As FPGAs have traditionally suffered from low programming models, accelerating applications on FPGAs has been a niche skill reserved to digital designers. While commercial C-to-gates high-level synthesis (HLS) tools have made great strides in their capabilities in the past decade, we show that such tools often fall short in key areas such as expressing and exploiting coarse-grained pipeline parallelism at multiple levels of granularity with arbitrary memory access patterns at each level. Producing efficient designs from HLS requires manually restructuring loops and providing various compiler hints in terms of pragmas, which requires hardware expertise. This dissertation argues that the real solution to FPGA programmability issues involves raising the level of programming abstraction beyond C. As parallel patterns represent the fundamental underpinnings of many domain-specific languages, starting from a parallel pattern description serves the dual purpose of raising the level of abstraction to the user as well as capturing rich semantic information about parallelism and locality to the compiler. A composable library of architectural templates called DHDL has been described, along with a compiler

flow that can efficiently generate hardware using the DHDL templates from parallel patterns. An evaluation of the compiler over an optimized baseline representative of optimized HLS designs shows speedups of up to 39.4×, with minimal impact on resource utilization. More importantly, the approach helps bridge the programmability chasm and helps democratize FPGA acceleration by providing a programming model accessible to software developers.

To address architectural overheads, this dissertation has described a new reconfigurable architecture called Plasticine. Motivated by the hardware templates, Plasticine consists of a hierarchical data path in Pattern Compute Units (PCUs) with pipelines of SIMD functional units to exploit fine-grained and coarse-grained parallelism. Distributed, programmable on-chip memories provided in Pattern Memory Units (PMUs), along with a decoupled access-execute model, exploits data locality and supports high bandwidth data access for various memory access patterns. Statically programmed interconnect switches allow communication of scalars, vectors, and control bits while minimizing area overhead. Dedicated address generators and scatter-gather units optimizes DRAM accesses for dense and sparse requests. Finally, a scalable control flow mechanism allows creating hierarchical pipelines of multiple levels while introducing minimal performance overhead. A thorough design space study that performs sensitivity analysis on various architectural parameters on area has been presented. Experiments performed to quantize the overheads of reconfiguration in Plasticine over specialized ASICs shows that the area overheads vary between 3× - 40× based on the application, with a geomean area overhead of 11.46× over dedicated ASICs. Plasticine has been implemented in RTL and synthesized with a 28nm technology library. Plasticine occupies an area of $113mm^2$, and consumes 49W of power at a 1GHz clock. Performance and energy evaluation of Plasticine over a Stratix V FPGA over a wide array of benchmarks shows that Plasticine can often outperform an FPGA, resulting in a speedup of up to 95× and a performance-per-Watt improvement of up to 77×.

Going forward, Plasticine can be viewed as a generic accelerator chip with variants that can be sized for specific domain-specific cost/performance points. In this respect, Plasticine should provide greater advantage over FPGAs as it allows customizing many parameters in the architecture beyond sizing, such as the datapath hierarchy, memory system, and interconnect. Plasticine can also be envisioned as a platform to co-design domain-specific

programmable or fixed-function ASICs with the associated software. Viewing Plasticine as a template, certain aspects of flexiblity can be traded off for performance to suit the needs of the domain. As a large fraction of ASIC development costs today goes towards software development [101], the co-design approach in Plasticine can provide a huge advantage; applications can be written using high-level constructs and compiled to the desired architecture several generations ahead of the actual hardware. This approach also provides necessary feedback between hardware and software early on in the design process, which helps making more informed hardware and software design decisions.

Bibliography

[1] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, Dec 2014.

[2] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 151–160, New York, NY, USA, 2014. ACM.

[3] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 255–268, New York, NY, USA, 2014. ACM.

[4] Eric S. Chung, John D. Davis, and Jaewon Lee. Linqits: Big data on little clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 261–272, New York, NY, USA, 2013. ACM.

[5] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263. IEEE, 2016.

[6] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *arXiv preprint arXiv:1602.01528*, 2016.

[7] Ardavan Pedram, Robert van de Geijn, and Andreas Gerstlauer. Codesign tradeoffs for high-performance, low-power linear algebra architectures. *IEEE Transactions on Computers, Special Issue on Power efficient computing*, 61(12):1724–1736, 2012.

[8] J. Dean, D. Patterson, and C. Young. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29, Mar 2018.

[9] Kunle Olukotun. Designing computer systems for software 2.0, isca 2018 keynote. `http://iscaconf.org/isca2018/docs/Kunle-ISCA-Keynote-2018.pdf`, 2018.

[10] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, Feb 2014.

[11] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[12] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498, New York, NY, USA, 2013. ACM.

[13] Dani Voitsechov and Yoav Etsion. Single-graph multiple flows: Energy efficient design alternative for gpgpus. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 205–216, Piscataway, NJ, USA, 2014. IEEE Press.

[14] Christopher J. Hughes. Single-instruction multiple-data execution. *Synthesis Lectures on Computer Architecture*, pages 1–121, 2015.

[15] Kenneth Czechowski, Victor W. Lee, Ed Grochowski, Ronny Ronen, Ronak Singhal, Richard Vuduc, and Pradeep Dubey. Improving the energy efficiency of big cores. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 493–504, Piscataway, NJ, USA, 2014. IEEE Press.

[16] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Tiark Rompf, and Kunle Olukotun. Locality-aware mapping of nested parallel patterns on gpus. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Micro, 2014.

[17] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[18] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. Accelerating deep convolutional neural networks using specialized hardware. Technical report, Microsoft Research, February 2015.

[19] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug 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*, pages 1–14, 2018.

[20] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. Sda: Software-defined accelerator for largescale dnn systems. Hot Chips 26, 2014.

[21] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Song Yao, Song Han, Yu Wang, and Huazhong Yang. From model to fpga: Software-hardware co-design for efficient neural network acceleration. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–27, Aug 2016.

[22] Amazon AWS. Amazon ec2 f1 instances. `https://aws.amazon.com/ec2/instance-types/f1`.

[23] David Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Queue*, 11(2):40:40–40:52, February 2013.

[24] The Khronos Group. OpenCL 2.0. `http://www.khronos.org/opencl/`.

[25] S.A. Edwards. The challenges of synthesizing hardware from c-like languages. *Design Test of Computers, IEEE*, 23(5):375–386, May 2006.

[26] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[27] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011.

[28] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, 2(2):135–253, February 2008.

[29] Benton. Highsmith Calhoun, Joseph F. Ryan, Sudhanshu Khanna, Mateja Putic, and John Lach. Flexible circuits and architectures for ultralow power. *Proceedings of the IEEE*, 98(2):267–282, Feb 2010.

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

[31] Kara K. W. Poon, Steven J. E. Wilton, and Andy Yan. A detailed power model for field-programmable gate arrays. *ACM Trans. Des. Autom. Electron. Syst.*, 10(2):279–302, April 2005.

[32] V. Adhinarayanan, I. Paul, J. L. Greathouse, W. Huang, A. Pattnaik, and W. Feng. Measuring and modeling on-chip interconnect power on real hardware. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–11, Sept 2016.

[33] R. Tessier, K. Pocek, and A. DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):332–354, March 2015.

[34] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix*, pages 61–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[35] Rainer Kress. A fast reconfigurable alu for xputers. 1996.

[36] Venkatraman. Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, Sept 2012.

[37] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. Piperench: A co/processor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 28–39, Washington, DC, USA, 1999. IEEE Computer Society.

[38] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai 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, pages 163–174, New York, NY, USA, 2006. ACM.

[39] Mingyu Gao and Christos Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–137, March 2016.

[40] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Triggered instructions: A control paradigm for spatially-programmed architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 142–153, New York, NY, USA, 2013. ACM.

[41] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 45:1–45:6, New York, NY, USA, 2017. ACM.

[42] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksander Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *European Conference on Object Oriented Programming*, ECOOP, 2013.

[43] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[44] M. Odersky. Scala. `http://www.scala-lang.org`, 2011.

[45] Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, feb 1999.

[46] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. In *TECS'14: ACM Transactions on Embedded Computing Systems*, July 2014.

[47] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, pages 47–56, New York, NY, USA, 2011. ACM.

[48] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, OSDI, pages 137–150, 2004.

[49] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[50] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. Spartan: A distributed array framework with smart tiling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 1–15, Santa Clara, CA, July 2015. USENIX Association.

[51] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 89–108, 2010.

[52] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs. POPL, 2013.

[53] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *International Symposium on Code Generation and Optimization,*, CGO, 2016.

[54] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI. ACM, 2010.

[55] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):24, 2013.

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

[57] Vivado design suite 2015.1 user guide: High-level synthesis.

[58] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 97–108. IEEE, 2014.

[59] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 651–665, New York, NY, USA, 2016. ACM.

[60] Ardavan Pedram, Stephen Richardson, Sameh Galal, Shahar Kvatinsky, and Mark Horowitz. Dark memory and accelerator-rich system optimization in the dark silicon era. *IEEE Design & Test*, 34(2):39–50, 2017.

[61] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 115–127, Piscataway, NJ, USA, 2016. IEEE Press.

[62] Maxeler Technologies. MaxCompiler white paper, 2011.

[63] H.M. Hussain, K. Benkrid, H. Seker, and A.T. Erdogan. Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 248–255, June 2011.

[64] Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. Optiml: an implicitly parallel domainspecific language for machine learning. In *in Proceedings of the 28th International Conference on Machine Learning, ser. ICML*, 2011.

[65] Andy Ye and Jonathan Rose. Using bus-based connections to improve field-programmable gate-array density for implementing datapath circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):462–473, May 2006.

[66] Intel Corporation. Intel stratix series 10 fpgas. `https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html`, 2017.

[67] David Lewis, Gordon Chiu, Jeffrey Chromczak, David Galloway, Ben Gamsa, Valavan Manohararajah, Ian Milton, Tim Vanderhoek, and John Van Dyken. The stratix^{TM}10 highly pipelined fpga architecture. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 159–168, New York, NY, USA, 2016. ACM.

[68] Jonathan. Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[69] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, Jan 2011.

[70] Ardavan Pedram, Andreas Gerstlauer, and Robert van de Geijn. On the efficiency of register file versus broadcast interconnect for collective communications in data-parallel hardware accelerators. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 19–26, 2012.

[71] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[72] Louis-Noël Pouchet. *Interative Optimization in the Polyhedral Model*. PhD thesis, University of Paris-Sud 11, Orsay, France, January 2010.

[73] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[74] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*, pages 549–562, Austin, TX, January 2011. ACM Press.

[75] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, Citeseer, 2008.

[76] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[77] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *ETAPS International Conference on Compiler Construction (CC'2010)*, pages 283–303, Paphos, Cyprus, March 2010. Springer Verlag.

[78] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 521–532, New York, NY, USA, 2015. ACM.

[79] Eric Hielscher. *Locality Optimization For Data Parallel Programs*. PhD thesis, New York University, 2013.

[80] Andrew R. Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. Chimps: A high-level compilation flow for hybrid cpu-fpga architectures. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 261–261, New York, NY, USA, 2008. ACM.

[81] Satnam Singh and David J. Greaves. Kiwi: Synthesis of fpga circuits from parallel programs. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '08, pages 3–12, Washington, DC, USA, 2008. IEEE Computer Society.

[82] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification invited talk. In *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '03, pages 249–, Washington, DC, USA, 2003. IEEE Computer Society.

[83] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. Autopilot: A platform-based esl synthesis system. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 99–112. Springer Netherlands, 2008.

[84] D. Petkov, R. Harr, and S. Amarasinghe. Efficient pipelining of nested loops: unroll-and-squash. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 6 pp–, April 2002.

[85] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *TACO*, 14(3):26:1–26:25, 2017.

[86] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. Elasticflow: A complexity-effective approach for pipelining irregular loop nests. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 78–85, Piscataway, NJ, USA, 2015. IEEE Press.

[87] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. Cgpa: Coarse-grained pipelined accelerators. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 78:1–78:6, New York, NY, USA, 2014. ACM.

[88] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the*

ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, pages 29–38, New York, NY, USA, 2013. ACM.

[89] Tim J. Todman, George A. Constantinides, Steve J. E. Wilton, Oskar Mencer, Wayne Luk, and Peter Y. K. Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings - Computers and Digital Techniques*, 152(2):193–207, Mar 2005.

[90] Reiner Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '01, pages 642–649, Piscataway, NJ, USA, 2001. IEEE Press.

[91] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The garp architecture and c compiler. *Computer*, 33(4):62–69, Apr 2000.

[92] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 422–433, New York, NY, USA, 2003. ACM.

[93] Paul Gratz, Changkyu Kim, Karthikeyan Sankaralingam, Heather Hanson, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. On-chip interconnection networks of the trips chip. *IEEE Micro*, 27(5):41–50, September 2007.

[94] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002.

[95] Darren C. Cronquist, Chris Fisher, Miguel Figueroa, Paul Franklin, and Carl Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*, pages 23–40, Mar 1999.

[96] Brian Van Essen, Aaron Wood, Allan Carroll, Stephen Friedman, Robin Panda, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Static versus scheduled interconnect in coarse-grained reconfigurable arrays. In *2009 International Conference on Field Programmable Logic and Applications*, pages 268–275, Aug 2009.

[97] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. The wavescalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4:1–4:54, May 2007.

[98] Brian Van Essen, Aaron Wood, Allan Carroll, Stephen Friedman, Robin Panda, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Static versus scheduled interconnect in coarse-grained reconfigurable arrays. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 268–275. IEEE, 2009.

[99] Grigoris Dimitroulakos, Michalis D Galanis, and Constantinos 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*, pages 10–pp. IEEE, 2006.

[100] Chris Nicol. A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing.

[101] Handel Jones. Strategies in optimizing market positions for semiconductor vendors based on ip leverage, ibs white paper. `http://www.ibs-inc.net`, 2014.