Plasticine: A Reconfigurable Architecture For Parallel Patterns

Raghu Prabhakar   Yaqi Zhang   David Koeplinger   Matt Feldman
Tian Zhao   Stefan Hadjis   Ardavan Pedram   Christos Kozyrakis   Kunle Olukotun

Stanford University
{raghup17,yaqiz,dkoeplin,mattfel,tianzhao,shadjis,perdavan,kozyraki,kunle}@stanford.edu

ABSTRACT

Reconfigurable architectures have gained popularity in recent years as they allow the design of energy-efficient accelerators. Fine-grain fabrics (e.g. FPGAs) have traditionally suffered from performance and power inefficiencies due to bit-level reconfigurable abstractions. Both fine-grain and coarse-grain architectures (e.g. CGRAs) traditionally require low level programming and suffer from long compilation times. We address both challenges with *Plasticine*, a new spatially reconfigurable architecture designed to efficiently execute applications composed of parallel patterns. Parallel patterns have emerged from recent research on parallel programming as powerful, high-level abstractions that can elegantly capture data locality, memory access patterns, and parallelism across a wide range of dense and sparse applications.

We motivate Plasticine by first observing 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. Based on these observations, we architect Plasticine as a collection of *Pattern Compute Units* and *Pattern Memory Units*. Pattern Compute Units are multi-stage pipelines of reconfigurable SIMD functional units that can efficiently execute nested patterns. Data locality is exploited in Pattern Memory Units using banked scratchpad memories and configurable address decoders. Multiple on-chip address generators and scatter-gather engines make efficient use of DRAM bandwidth by supporting a large number of outstanding memory requests, memory coalescing, and burst mode for dense accesses. 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 a cycle-accurate simulator, we demonstrate that 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.

CCS CONCEPTS

• **Hardware** → **Hardware accelerators**; • **Software and its engineering** → *Retargetable compilers*;

KEYWORDS

parallel patterns, reconfigurable architectures, hardware accelerators, CGRAs

1 INTRODUCTION

In the search for higher performance and energy efficiency, computing systems are steadily moving towards the use of specialized accelerators [7, 9–11, 19, 33, 44]. 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. This makes ASIC accelerators impractical for all but the most ubiquitous applications.

Reconfigurable architectures like FPGAs offset the high NRE fabrication costs by providing flexible logic blocks in a statically programmable interconnect to implement custom 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. This flexibility has resulted in a number of successful commercial FPGA-based accelerators deployed in data centers [28, 29, 37]. However, flexibility comes at the cost of architectural inefficiencies. Bit-level reconfigurability in computation and interconnect resources comes with significant area and power overheads. For example, over 60% of the chip area and power in an FPGA is spent in the programmable interconnect [4, 5, 22, 35]. Long combinational paths through multiple logic elements limit the maximum clock frequency at which an accelerator design can operate. These inefficiencies have motivated the development of coarse-grain reconfigurable architectures (CGRAs) with word-level functional units that match the compute needs of most accelerated applications. CGRAs provide dense compute resources, power efficiency, and clock frequencies up to an order of magnitude higher than FPGAs. 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.

Map	FlatMap

Fold	HashReduce
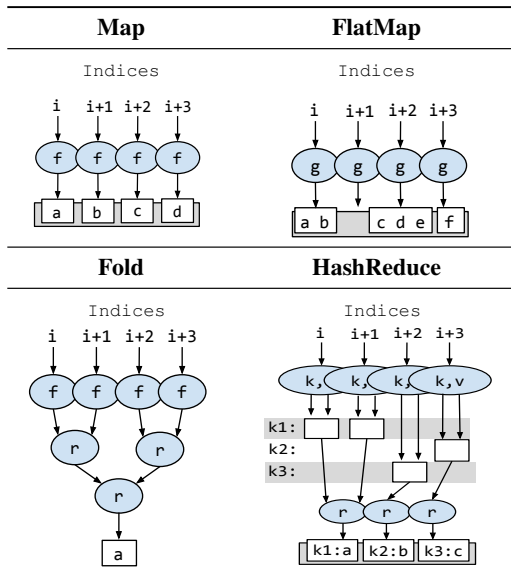
Table 1: The parallel patterns in our programming model.

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.

Unfortunately, both FPGAs and previously proposed CGRAs are difficult to use. Accelerator design typically involves low-level programming models and long compilation times [3, 21, 22]. The heterogeneity of resources in most CGRAs and in FPGAs with coarse-grain blocks adds further complications. A promising approach towards simplifying accelerator development is to start with domain-specific languages that capture high-level parallel patterns such as map, reduce, filter, and flatmap [38, 41]. Parallel patterns have been successfully used to simplify parallel programming and code generation for a diverse set of parallel architectures including multi-core chips [27, 34, 40] and GPUs [8, 23]. Recent work has shown that parallel patterns can also be used to generate optimized accelerators for FPGAs from high-level languages [15, 36]. In this work, we focus 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.

We introduce *Plasticine*, a new spatially reconfigurable accelerator architecture optimized for efficient execution of parallel patterns. Plasticine is a two dimensional array of two kinds of coarse-grained reconfigurable units: *Pattern Compute Units* (PCUs) and *Pattern Memory Units* (PMUs). Each PCU consists of a reconfigurable pipeline with multiple stages of SIMD functional units, with support for cross-SIMD lane shifting and reduction. PMUs are composed of a banked scratchpad memory and dedicated addressing logic and address decoders. These units communicate with each other through a pipelined *static hybrid interconnect* with separate bus-level and word-level data, and bit-level control networks. The hierarchy in the Plasticine architecture simplifies compiler mapping and improves execution efficiency. The compiler can map inner loop computation to one PCU such that most operands are transferred directly between

functional units without scratchpad accesses or inter-PCU communication. The on-chip, banked scratchpads are configurable to support streaming and double buffered accesses. The off-chip memory controllers support both streaming (burst) patterns and scatter/gather accesses. Finally, the on-chip control logic is configurable to support nested patterns.

We have implemented Plasticine in Chisel [2], a Scala-based hardware definition language. We obtain area estimates after synthesizing the design using Synopsys Design Compiler, and power numbers using simulation traces and PrimeTime. Using VCS and DRAM-Sim2 for cycle-accurate simulation, we perform detailed evaluation of the Plasticine architecture on a wide range of dense and sparse benchmarks in the domains of linear algebra, machine learning, data analytics and graph analytics.

The rest of this paper is organized as follows: Section 2 reviews the key concepts in parallel patterns and their hardware implementation. Section 3 introduces the Plasticine architecture and explores key design tradeoffs. Section 4 evaluates the power and performance efficiency of Plasticine versus an FPGA. Section 5 discusses related work.

2 PARALLEL PATTERNS
2.1 Programming with 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 [38, 41]. Previous work has shown how parallel patterns can be used in functional programming models to generate multi-threaded C++ for CPUs comparable to hand optimized code [40] and efficient accelerator designs for FPGAs [1, 15, 36]. As with FPGAs and multi-core CPUs, knowledge of data parallelism is vital to achieve good performance when targeting CGRAs. This implicit knowledge makes parallel patterns a natural programming model to drive CGRA design.

Like previous work on hardware generation from parallel patterns [15, 36], our programming model is based on the parallel patterns *Map*, *FlatMap*, *Fold*, and *HashReduce*. These patterns are selected because they are most amenable to hardware acceleration. Table 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. Each of these patterns builds an output and reads from an arbitrary number of input collections.

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

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

```
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.hashReduce{ 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: Example of using filter (FlatMap) and HashReduce in a Scala-based language, inspired by TPC-H query 1.

of a gather, a standard element-wise map, a zip, a windowed filter, or any combination thereof.

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*.

HashReduce 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*. HashReduce 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 HashReduce where the *key* function gives the histogram bin, the *value* function is defined to always be "1", and the *combine* function is integer addition.

Figure 1 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

	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 2: Programming model components and their corresponding hardware implementation requirements.

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 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 HashReduce with an index domain with the size of the before collection.

2.2 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 [8, 36, 38, 41]. 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 2.

First, all four patterns 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 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.

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. Previous work has shown how to tile parallel patterns to introduce statically sized windows of reuse into the application and potentially increase data locality [36].

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 [32]. 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 architecture 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.

While many coarse-grained hardware accelerators have been proposed, no single accelerator described by previous work has all of these hardware features. This means that, while some of these accelerators can be targeted by parallel patterns, none of them can fully exploit the properties of these patterns to achieve maximum performance. Traditional FPGAs can also be configured to implement these patterns, but with much poorer energy efficiency, as we show in Section 4. We discuss related work further in Section 5.

3 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 3.7

3.1 Pattern Compute Unit

The PCU is designed to execute a single, innermost parallel pattern in an application. As shown in Figure 3, 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 (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.

3.2 Pattern Memory Unit

Figure 4 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
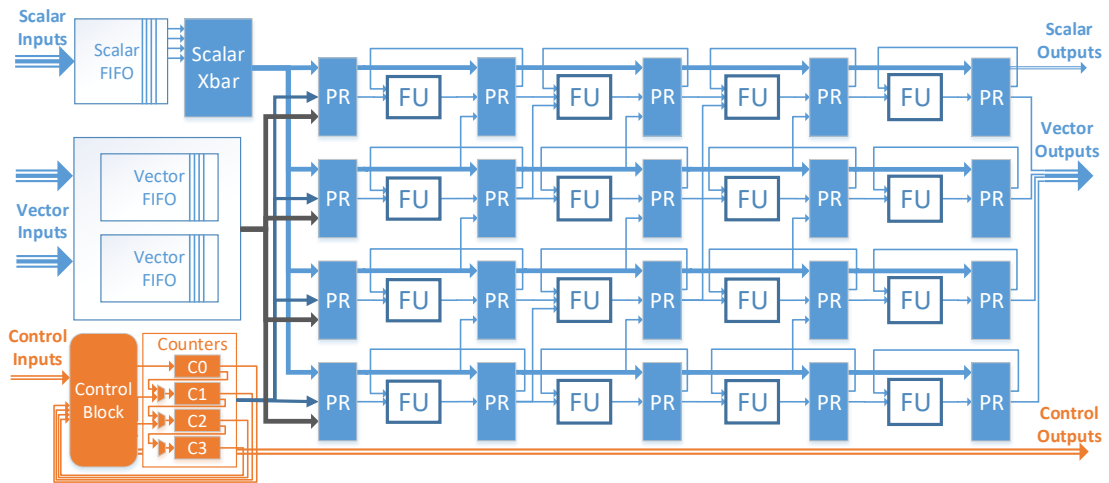
Figure 3: Pattern Compute Unit (PCU) architecture. We show only 4 stages and 4 SIMD lanes, and omit some control signals.
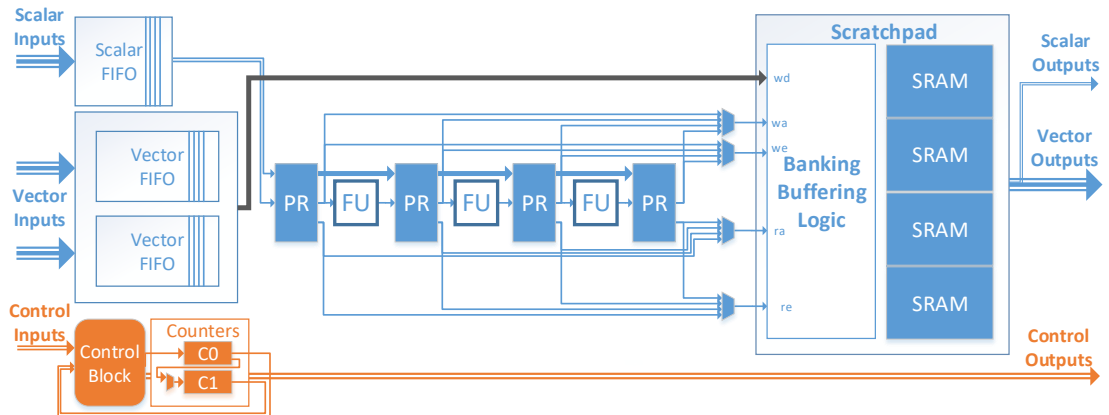

Figure 4: Pattern Memory Unit (PMU) architecture: configurable scratchpad, address calculation datapath, and control.
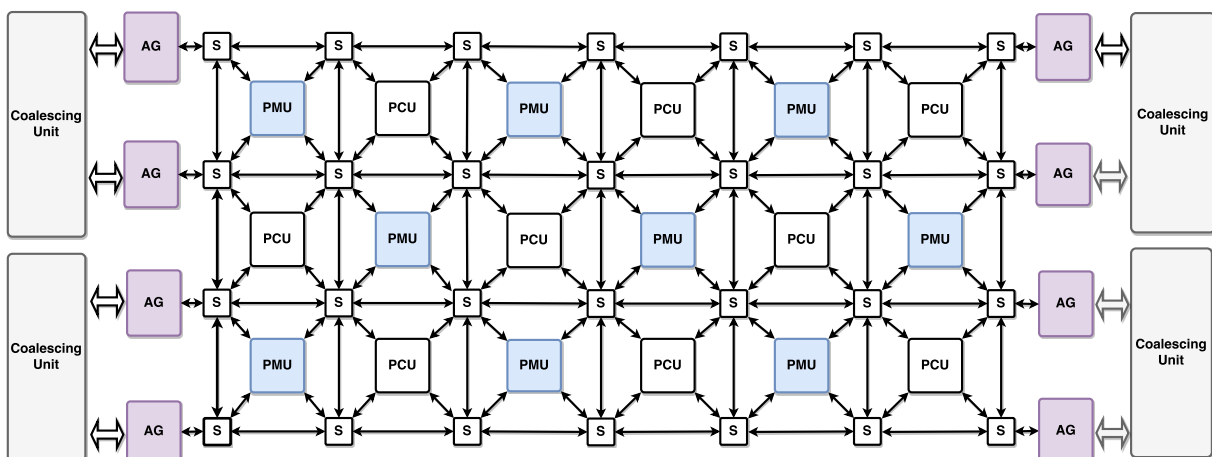

**Figure 5: Plasticine chip-level architecture (actual organization 16 x 8). All three networks have the same structure.
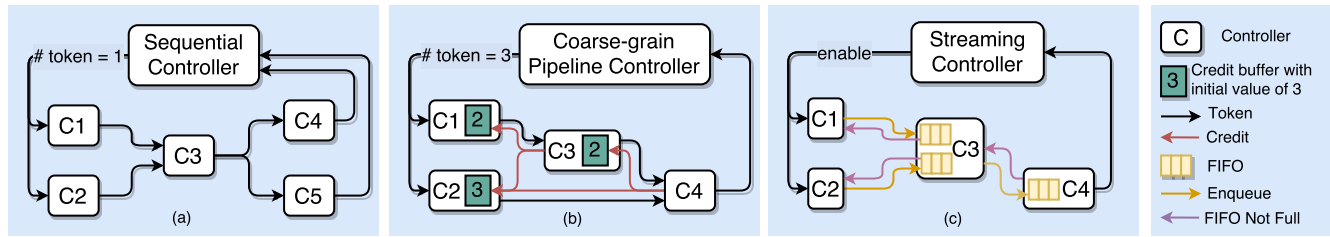PCU:Pattern Compute Unit, PMU: Pattern Memory Unit, AG: Address Generator, S: Switch Box.**

Figure 6: Sequential, coarse-grained pipelining, and streaming control schemes.

Figure 5, 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.

3.3 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 5. 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.

3.4 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 5. Each AG contains a reconfigurable scalar datapath to generate DRAM requests, similar in structure to the PMU datapath shown in Figure 4. 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.

3.5 Control Flow

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. 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 6). 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.

3.6 Application Mapping

We begin with an application represented as a hierarchy of parallelizable dataflow pipelines written in a parallel pattern-based language called the Delite Hardware Definition Language (DHDL) [20]. Prior work [36] has shown how applications expressed in the parallel patterns described in Section 2 can be automatically decomposed into pipelines in DHDL. 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 3.5. 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.

3.7 Architecture Sizing

Thus far, we have described a parameterized architecture composed of, among other things, PCUs, PMUs, and interconnect. We now
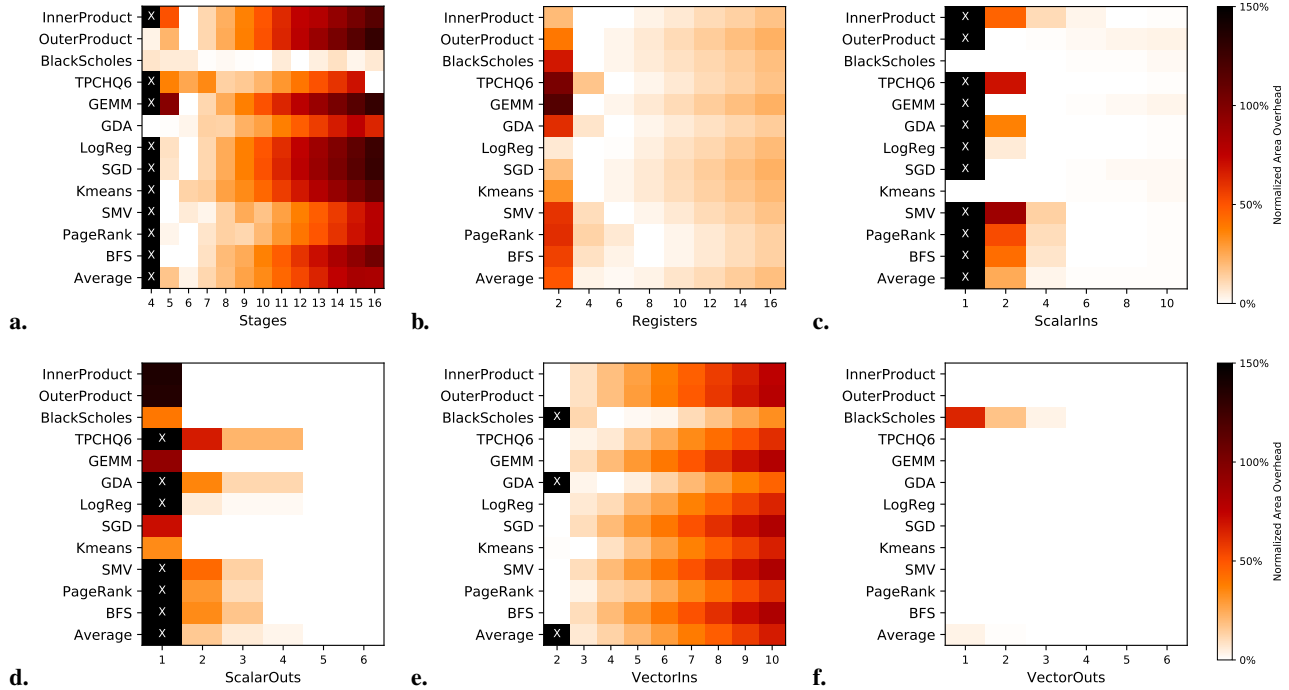
Figure 7: Area overhead ($Area_{PCU}/Min_{PCU} - 1$) while sweeping various Plasticine PCU parameters for a subset of our benchmarks. Min_{PCU} **is benchmark-specific minimum possible area. Areas marked with an $\times$ denote invalid parameters for the given application.** *a.* **Stages per PCU;** *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.**

	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 3: Design space and final selected parameters.

describe our process for tuning the PCU and PMU parameters to create the final Plasticine architecture that we evaluate in Section 4. Table 3 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. 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 3.6.

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 other parameters are left unrestricted. Figure 7a 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 7b, 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 7(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

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: Evaluation benchmarks.

therefore want to minimize vector inputs as much as possible. However, as seen in Figure 7e, due to limitations in splitting across PCUs, BlackScholes and GDA are restricted to having at least 3 vector inputs. Figure 7f 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 3. 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 16×8 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 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.

4.1 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 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 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 the Delite Hardware Definition Language (DHDL), a specialized language based on parallel patterns for writing applications for spatial architectures [20]. In DHDL, applications are specified as hierarchies of parallelizable dataflow pipelines. Previous work [36] has shown that DHDL can be automatically generated from parallel patterns, and can be used to generate efficient hardware accelerator designs for FPGAs [20].

4.2 Plasticine Design

We implement the Plasticine architecture in Chisel [2] using the selected parameters listed in Table 3. 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 5 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 DRAM-Sim2 [39] 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 3.6. 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).

	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 5: Plasticine area breakdown.

4.3 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 6) 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 3, including configuration logic, configurable banked memories, and statically configurable ALUs. Column *a.* of Table 6 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 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 6, 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

Benchmark	Coarse	Homogeneous				Generalized			
	a.	*b.* PMUs		*c.* PCUs		*d.* PMUs		*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 6: 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.

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 under-utilize 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 3.7, 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 [33], 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 3.7. 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 7, 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.

4.4 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.

4.5 Plasticine versus FPGA

Table 7 shows the utilization, 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 7 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 7, we can see that using 65% of PCUs, Black-Scholes maximizes DRAM bandwidth utilization, achieving a speedup of 5.1×.

| Benchmark | Utilization (%) | | | | | | | Power (W) | | Plasticine / FPGA | | |
| | FPGA | | Plasticine | | | | | FPGA | Plasticine | | | |
	Logic	Memory	PCU	PMU	AG	FU	Register			Power	Performance	Perf/W
Inner Product	24.3	33.5	17.2	25.0	47.1	69.8	10.2	21.8	18.9	0.9	1.4	1.6
Outer Product	38.2	71.4	15.6	46.9	88.2	21.6	12.8	24.4	26.9	1.1	6.7	6.1
Black-Scholes	68.9	100.0	65.6	21.9	41.2	25.1	53.4	28.3	24.7	0.9	5.1	5.8
TPCH-Query 6	24.3	33.4	28.1	25.0	47.1	70.8	20.2	21.7	20.5	0.9	1.4	1.5
GEMM	40.4	94.8	34.4	68.8	97.1	56.0	8.6	25.6	34.6	1.4	33.0	24.4
GDA	53.6	96.8	89.1	87.5	44.1	8.1	11.2	26.5	41.0	1.5	40.0	25.9
LogReg	28.4	73.4	51.6	68.8	8.8	30.2	12.3	22.9	28.6	1.2	11.4	9.2
SGD	60.1	58.2	6.3	9.4	8.8	34.3	7.2	25.6	10.7	0.4	6.7	15.9
Kmeans	42.1	65.4	10.9	17.2	8.8	35.5	10.9	23.9	12.9	0.5	6.1	11.3
CNN	86.8	99.0	48.9	98.4	100.0	62.5	25.0	34.4	42.6	1.2	95.1	76.9
SMDV	27.3	31.0	43.8	15.6	29.4	10.4	2.7	21.5	19.3	0.9	8.3	9.3
PageRank	31.3	33.4	28.1	20.3	20.6	3.9	1.9	21.9	17.1	0.8	14.2	18.2
BFS	25.3	45.9	18.8	15.6	11.8	3.1	1.5	21.9	14.0	0.6	7.3	11.4

Table 7: Resource utilization, power, performance, and performance-per-Watt comparisons between Plasticine and FPGA.

GEMM and GDA are compute-bound, with ample temporal and spatial locality. On Plasticine, GEMM achieves a speedup of $33.0\times$. GDA performs similarly with a speedup of $40.0\times$. 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 [31].

CNN is another compute-intensive benchmark where Plasticine outperforms the FPGA, in this case by $95.1\times$. 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 $11.4\times$ 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 $6.7\times$ and $6.1\times$ 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 $39.8\times$ and $12.3\times$ respectively.

SMDV, PageRank, and BFS achieve speedups of $8.3\times$, $14.2\times$, and $7.3\times$ 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.

5 RELATED WORK

Table 2 introduced key architectural features required to efficiently execute parallel patterns. We now discuss the significant related work associated with these features.

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 [25], DySER [17], Garp [6], and Tartan [26] 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 [16] 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 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 [16].

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 [18] and about 30% of the total dynamic power on the GPU [24]. Furthermore, using a reconfigurable datapath in place of a conventional instruction pipeline in a GPU reduces energy consumption by about 57% [43]. The Raw microprocessor [42] 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.

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 [12] is a one-dimensional array of ALUs, registers, and memories with hardware support for static and dynamic control. A subsequent research project called Mosaic [13] 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 [14] 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 [30] 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.

6 CONCLUSION

In this paper we describe Plasticine, 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.

REFERENCES

[1] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. 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)*. 89–108. https://doi.org/10.1145/1869459.1869469

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

[3] David Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses. *Queue* 11, 2, Article 40 (Feb. 2013), 13 pages. https://doi.org/10.1145/2436696.2443836

[4] Ivo Bolsens. 2006. Programming Modern FPGAs, International Forum on Embedded Multiprocessor SoC, Keynote,. http://www.xilinx.com/univ/mpsoc2006keynote.pdf.

[5] Benton. Highsmith Calhoun, Joseph F. Ryan, Sudhanshu Khanna, Mateja Putic, and John Lach. 2010. Flexible Circuits and Architectures for Ultralow Power. *Proc. IEEE* 98, 2 (Feb 2010), 267–282. https://doi.org/10.1109/JPROC.2009.2037211

[6] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. 2000. The Garp architecture and C compiler. *Computer* 33, 4 (Apr 2000), 62–69. https://doi.org/10.1109/2.839323

[7] Jared Casper and Kunle Olukotun. 2014. Hardware Acceleration of Database Operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA '14)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/2554688.2554787

[8] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2011. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP)*. ACM, New York, NY, USA, 47–56. https://doi.org/10.1145/1941553.1941562

[9] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 609–622. https://doi.org/10.1109/MICRO.2014.58

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

[11] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits: Big Data on Little Clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 261–272. https://doi.org/10.1145/2485922.2485945

[12] Darren C. Cronquist, Chris Fisher, Miguel Figueroa, Paul Franklin, and Carl Ebeling. 1999. Architecture design of reconfigurable pipelined datapaths. In *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*. 23–40. https://doi.org/10.1109/ARVLSI.1999.756035

[13] Brian Van Essen, Aaron Wood, Allan Carroll, Stephen Friedman, Robin Panda, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. 2009. Static versus scheduled interconnect in Coarse-Grained Reconfigurable Arrays. In *2009 International Conference on Field Programmable Logic and Applications*. 268–275. https://doi.org/10.1109/FPL.2009.5272293

[14] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 126–137. https://doi.org/10.1109/HPCA.2016.7446059

[15] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. 2014. Hardware system synthesis from Domain-Specific Languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. 1–8. https://doi.org/10.1109/FPL.2014.6927454

[16] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. 1999. PipeRench: A Co/Processor for Streaming Multimedia Acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA '99)*. IEEE Computer Society, Washington, DC, USA, 28–39. https://doi.org/10.1145/300979.300982

[17] Venkatraman. Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro* 32, 5 (Sept 2012), 38–51. https://doi.org/10.1109/MM.2012.51

[18] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding Sources of Inefficiency in General-purpose Chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 37–47. https://doi.org/10.1145/1815961.1815968

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

[20] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *International Symposium in Computer Architecture*.

[21] Ian Kuon and Jonathan Rose. 2007. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (Feb 2007), 203–215. https://doi.org/10.1109/TCAD.2006.884574

[22] Ian Kuon, Russell Tessier, and Jonathan Rose. 2008. FPGA Architecture: Survey and Challenges. *Found. Trends Electron. Des. Autom.* 2, 2 (Feb. 2008), 135–253. https://doi.org/10.1561/1000000005

[23] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Tiark Rompf, and Kunle Olukotun. 2014. Locality-Aware Mapping of Nested Parallel Patterns on GPUs. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (IEEE Micro)*.

[24] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 487–498. https://doi.org/10.1145/2485922.2485964

[25] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix*. Springer Berlin Heidelberg, Berlin, Heidelberg, 61–70. https://doi.org/10.1007/978-3-540-45234-8_7

[26] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. 2006. 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)*. ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/1168857.1168878

[27] M. Odersky. 2011. Scala. http://www.scala-lang.org. (2011).

[28] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. 2014. SDA: Software-Defined Accelerator for LargeScale DNN Systems *(Hot Chips 26)*.

[29] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. 2015. *Accelerating Deep Convolutional Neural Networks Using Specialized Hardware*. Technical Report. Microsoft Research. http://research-srv.microsoft.com/pubs/240715/CNN%20Whitepaper.pdf

[30] 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. 2013. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 142–153. https://doi.org/10.1145/2485922.2485935

[31] Ardavan Pedram, Andreas Gerstlauer, and Robert van de Geijn. 2012. 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)*. 19–26. https://doi.org/10.1109/SBAC-PAD.2012.35

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

[33] Ardavan Pedram, Robert van de Geijn, and Andreas Gerstlauer. 2012. Codesign Tradeoffs for High-Performance, Low-Power Linear Algebra Architectures. *IEEE Transactions on Computers, Special Issue on Power efficient computing* 61, 12 (2012), 1724–1736.

[34] 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. 1999. Haskell 98 — A Nonstrict, Purely Functional Language. Available from http://www.haskell.org/definition/. (feb 1999).

[35] Kara K. W. Poon, Steven J. E. Wilton, and Andy Yan. 2005. A Detailed Power Model for Field-programmable Gate Arrays. *ACM Trans. Des. Autom. Electron. Syst.* 10, 2 (April 2005), 279–302. https://doi.org/10.1145/1059876.1059881

[36] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. 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)*. ACM, New York, NY, USA, 651–665. https://doi.org/10.1145/2872362.2872415

[37] 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. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[38] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. 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)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[39] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[40] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. In *TECS'14: ACM Transactions on Embedded Computing Systems*.

[41] 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. 2013. Composition and Reuse with Compiled Domain-Specific Languages. In *European Conference on Object Oriented Programming (ECOOP)*.

[42] 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. 2002. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro* 22, 2 (March 2002), 25–35. https://doi.org/10.1109/MM.2002.997877

[43] Dani Voitsechov and Yoav Etsion. 2014. Single-graph Multiple Flows: Energy Efficient Design Alternative for GPGPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 205–216. http://dl.acm.org/citation.cfm?id=2665671.2665703

[44] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. 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)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/2541940.2541961