AFRL-RY-WP-TR-2020-0009

PLASTICINE – A UNIVERSAL DATA ANALYTICS ACCELERATOR

Kunle Olukotun
Leland Stanford Junior University

MARCH 2020
Final Report

STINFO COPY

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with The Under Secretary of Defense memorandum dated 24 May 2010 and AFRL/DSO policy clarification email dated 13 January 2020. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2020-0009 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

*//Signature//

KERRY L. HILL
Program Manager
Sensor Subsystems Branch
Aerospace Components & Subsystems Division

//Signature//

TIMOTHY R. JOHNSON, Chief
Sensor Subsystems Branch
Aerospace Components & Subsystems Division

//Signature//

ADAM L. BROOKS, Lt Col, USAF
Deputy
Aerospace Components & Subsystems Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
March 2020	Final	30 July 2018 – 30 September 2019

4. TITLE AND SUBTITLE
PLASTICINE – A UNIVERSAL DATA ANALYTICS ACCELERATOR

5a. CONTRACT NUMBER
FA8650-18-2-7865

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62716E

6. AUTHOR(S)
Kunle Olukotun

5d. PROJECT NUMBER
N/A

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
Y1UN

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Leland Stanford Junior University
450 Serra Mall
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

Defense Advanced
Research Projects Agency
DARPA/MTO
675 North Randolph Street
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYDR

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2020-0009

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with The Under Secretary of Defense memorandum dated 24 May 2010 and AFRL/DSO policy clarification email dated 13 January 2020. This material is based on research sponsored by Air Force Research laboratory (AFRL) and the Defense Advanced Research Agency (DARPA) under agreement number FA8650-18-2-7865. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation herein. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies of endorsements, either expressed or implied, of AFRL and DARPA or the U.S. Government. Report contains color.

14. ABSTRACT
We have developed hardware and software for a universal data analytics accelerator called Plasticine. Plasticine hardware is based on the novel concept of a reconfigurable dataflow architecture (RDA) which has both reconfigurable memories and reconfigurable compute. RDAs provide high energy efficiency without sacrificing programmability. We have fabricated a 7nm chip implementation of Plasticine I that provides significant performance and energy improvements compared to GPUs and FPGAs. Architecture studies for Plasticine II include support for dynamic on-chip networks, sparse-matrix computations and graph analytics. Plasticine software includes high-level and low-level compilers for converting TensorFlow machine learning applications into optimized configurations for Plasticine.

15. SUBJECT TERMS
machine learning accelerators, reconfigurable dataflow architectures, reconfigurable dataflow accelerators, optimizing compilers, spatial compilation, coarse-grain reconfigurable arrays, Plasticine chip fabrication, TensorFlow compilation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Kerry Hill
Unclassified	Unclassified	Unclassified	SAR	43	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Table of Contents

Section **Page**

List of Figures

Figure **Page**

1. SUMMARY

The overall goal of the Plasticine Software Defined Hardware (SDH) project is to develop a universal data analytics accelerator with support for dynamic reconfiguration based on our previous work on domain specific languages and parallel-pattern based Plasticine architecture. The original proposal called for the development of two prototype architectures Plasticine I and Plasticine II. The focus of Plasticine I is to actually implement an improved version of the current Plasticine static coarse grain reconfigurable array (CGRA) with the assistance of SambaNova Systems to serve as a vehicle for TA-2 software development and architecture exploration for Plasticine II using real workloads. Plasticine II was to include full support for dynamic reconfiguration. The original plan was to develop implementation for Plasticine I, develop the initial version of the software stack and perform the architecture exploration for Plasticine II during Phase 1 of SDH and then finish the development of the software and hardware during Phases 2 and 3 of the SDH program. Since we were not awarded funding for Phase 2, this report covers the initial software development, the architecture explorations for Plasticine II and the commercial implementation of Plasticine I by SambaNova Systems.

2. INTRODUCTION

The research described in this report builds on our previous work on data analytics (EmptyHeaded) and machine learning algorithms (Hogwild! and Buckwild!), high-level language compilers (Delite), and reconfigurable architectures (Spatial and Plasticine).

On this basis, we have developed a full-stack solution to SDH with the following features: high-level and low-level compiler optimizations based on parallel patterns; a highly-optimized way of dynamically mapping parallel patterns to a reconfigurable processing architecture composed of dynamically configurable pipelines and memories; and support for efficient sparse and dense off-chip memory access. The architecture enables the dynamic reconfiguration of hierarchical parallelism, physical layout and logical format of on-chip and off-chip memories, and precision to achieve superior performance on data-intensive algorithms. The reconfigurable processor is implemented and integrated into a system. By the end of the SDH program, this project will result in a commercially available SDH system with the programmability of a central processing unit (CPU) and similar energy efficiency and performance of an application-specific integrated circuit (ASIC).

Our processor architecture builds upon the Plasticine CGRA architecture (Prabhakar, et al. 2017) which provides improvements of up to two orders of magnitude over a field-programmable gate array (FPGA) in performance and performance-per-Watt on data-intensive algorithms. Plasticine embodies many of the properties outlined in the TA-1 architecture requirements, such as a coarse-grained datapath, a flexible on-chip memory system that can be reconfigured to suit various access patterns, an off-chip memory access system that efficiently supports both dense and sparse data accesses, and most importantly, it is an ideal compiler target for high-level data analytic applications.

In this report, we present our accomplishments during Phase 1 of Plasticine II development and the projected features of Plasticine II that were intended for Phase 2 and beyond. Our goal is to enable users to program in multiple paradigms, including databases, dense neural networks, sparse applications, and graph analytics, without sacrificing efficiency. To improve developer productivity, we introduce a new compiler stack that maps programs written in high level domain-specific languages (DSLs) to Plasticine II, with Spatial as an intermediate representation. To improve efficiency, we extend the original Spatial with new optimizations for more general parallel patterns shared across multiple domains, including new patterns like filter, and flatMap. This more general set of parallel patterns further drives the architectural design of Plasticine II. We detail these extensions in Section 3.2.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 TA2: Software and the Compiler Stack

In this section, we describe the existing compiler at the end of Phase 1 with various optimization performed at different levels of the stack. The overall Compiler flow is show in Figure 1.

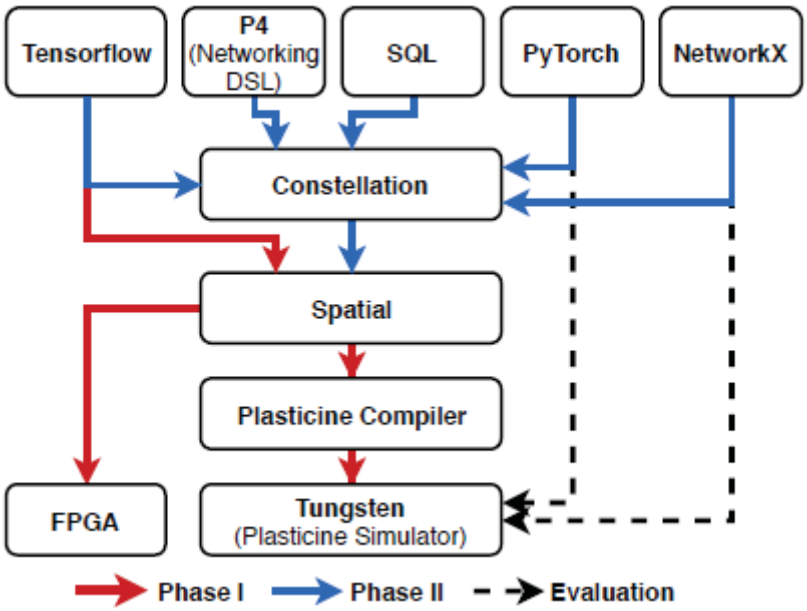

Figure 1: An overview of our SDH Compiler Stack

3.1.1 TensorFlow to Spatial Compiler

To improve developer productivity, we developed an end-to-end, open-source compiler that maps TensorFlow to Spatial (Hadjis, and Olukotun 2019). The compiler is capable of running state-of-the-art deep neural networks (DNNs) specified in TensorFlow for applications ranging from object recognition to speech-to-text translation. The compiler flow is summarized in Figure 2. A challenge in mapping machine learning (ML) models expressed in high-level frameworks to programmable hardware is that optimizations are needed at multiple abstraction levels. The compiler leverages both TensorFlow and Spatial to perform optimizations at multiple abstraction levels and deploy these high-level ML graphs to hardware. In addition to allowing developers to synthesize accelerators from a high-level framework, the compiler is also a hardware architecture exploration tool. Spatial's flexibility allows experimentation with various architecture choices (Hadjis, and Olukotun 2019). This allows developers to explore design space decisions, including accelerator operation granularity, hardware specialization, algorithm transformation, tensor storage format, and memory-level parallelism. To do this, the compiler relies on Spatial's ability to synthesize efficient hardware for many types of circuits, discussed in detail in the next section.

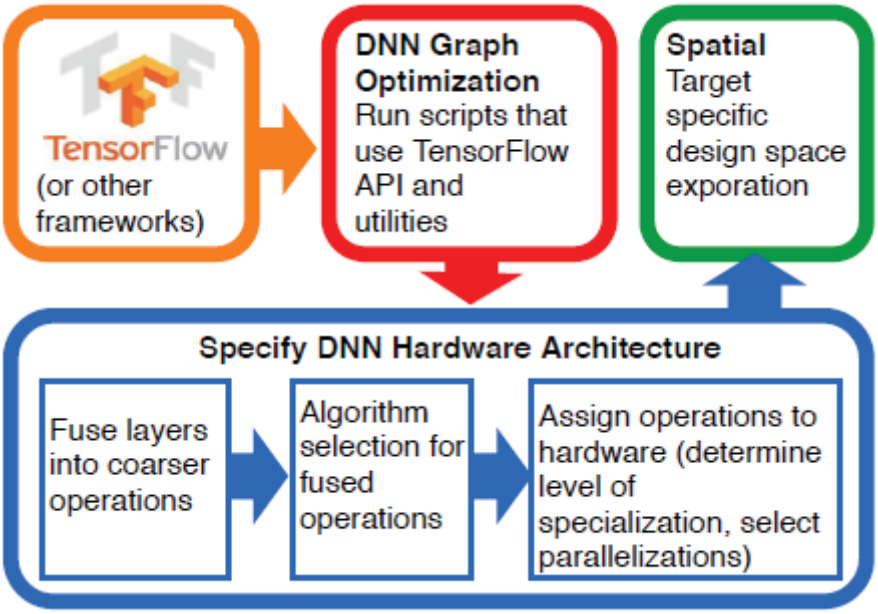

Figure 2: TensorFlow to Spatial Compiler

3.1.2 Spatial

Spatial is a hardware-centric DSL that expresses applications using parallel patterns. Unlike most software languages, Spatial has an explicit memory hierarchy, including syntax for transfers (dense and sparse) between dynamic random-access memory (DRAM) and on-chip memories. To speed up programs, Spatial allows the programmer to parallelize loops and employ scheduling directives dictating how loops execute relative to one another. It also automatically retimes pipelined operations and detects common reduction patterns that can be optimized with specialized hardware. To support these optimizations, Spatial analyzes memory access patterns, bank, and buffer on-chip memories to scale in on-chip read and write bandwidth. To scale on-chip memory bandwidth, Spatial analyzes memory access patterns and uses the results to bank and buffer on-chip memories.

Accumulation Optimizations: Spatial can detect a variety of accumulation patterns referencing non-addressable memories (i.e., `Registers`) and transform the infrared (IR) to optimize these cases. However, accumulations into addressable memories could require the compiler to throttle performance for guaranteed correctness. Consider the accumulation `sram(f(i)) = g(sram(h(i)))` inside a loop body.

With the loop-carried dependency, the trivial solution to guarantee correctness is to stall each iteration until the previous iteration has updated the memory. However, there are many cases where this is too conservative, including low-rank accumulations and regional accumulations. Instead, Spatial pipelines different iterations of the loop by analyzing the relationship between `f(i)` and `h(i)`, the range of iterator `i`, and the complexity of function `g`. Spatial contains an analysis pass for accumulations into addressable memory, determining the iteration space and computing the minimum number of iterations the controller must wait before it can issue the next iteration (i.e., the initialization interval).

Banking Optimizations: For each on-chip memory, Spatial analyzes all accesses and uses the controller hierarchy to identify which accesses must be time-multiplexed and which are bankable. If the concurrent accesses are derived from loop unrolling, the compiler determines a banking assignment for each unrolled lane as an affine function of the unrolled loop iterators. Next, the compiler tests the validity of the memory partitioning using conflict polytope emptiness testing. At runtime, accesses from different groups are active at different times. Each concurrent access within the group carries a bank ID and an offset within each bank that gets dynamically resolved. Figure 3 shows two groups of time-multiplexed accesses, each with four parallel accesses.

In cases when the banking IDs (b0-b7) can be statically resolved, the compiler eliminates the crossbar.

Spatial handles a variety of access patterns that can contain both affine and non-affine components. It can also bank for concurrent accesses, as long as it detects synchronization guarantees on the iterators.

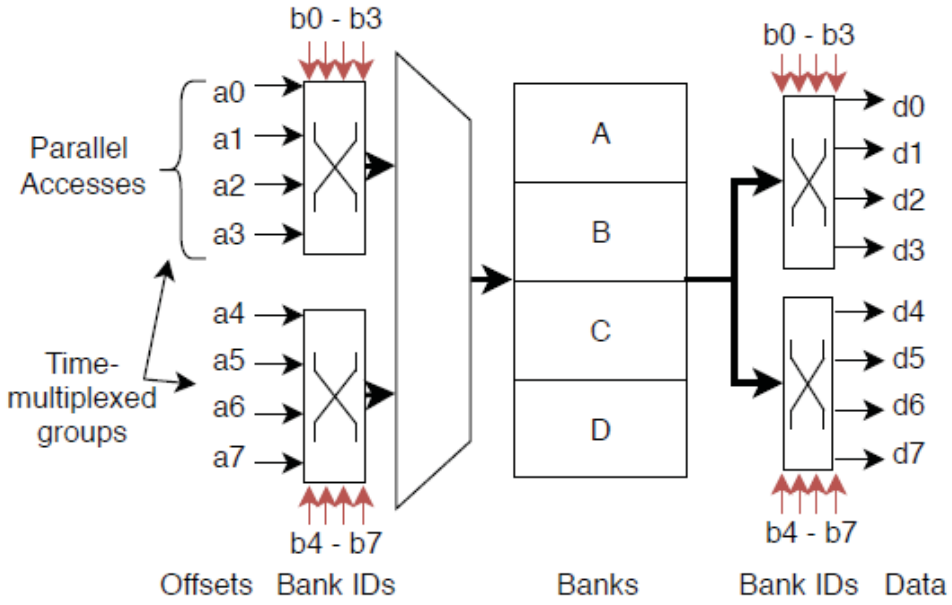

Figure 3: On-chip Memory Banking

Banking Cost Models: Although many memories can be parallelized using a variety of banking schemes, some schemes may be more hardware-efficient than others. Spatial can compute multiple candidate schemes and choose the best one. By using an ensemble of decision-tree based models and deep learning models, it computes the cost of both the hardware used to realize the banking scheme and the auxiliary nodes inserted into the address path to handle bank resolution. This provides a more accurate estimate of the cost than heuristics like bank switching and partition volumes.

Banking Report and Annotations: The compiler generates a report showing each memory's valid banking schemes and their estimated cost; the user can sort by estimated cost or compiler search time. The former allows the programmer to understand which accesses consume the most

resources, while the latter allows the programmer to see which memories the compiler had the most trouble banking. The programmer can use this report to either refactor these accesses in ways that result in less expensive banking schemes or select from a variety of banking annotations that steer the compiler towards the correct banking scheme more quickly. Understanding which accesses consume the most hardware resources allows the user to refactor programs to use less expensive banking schemes. Similarly, knowing which memories the compiler struggles to bank allows the programmer to add annotations that guide the compiler toward the correct banking scheme more quickly. Because Spatial spends most of its time analyzing banking, addressing these issues decreases compilation time significantly.

Do-While Solution: Many sparse and graph applications employ dynamic, data- dependent control flow patterns, like "run until converged" and "process an unknown number of incoming packets." Although these map well to procedural software, they are tricky to implement in hardware. Spatial has Do-While loops that capture these situations; these loops are implemented using infinite counter chains and explicit break conditions. Spatial also allows users to implement arbitrary Finite State Machines (FSMs) to capture non-linear iteration spaces.

Design Space Exploration for Reconfigurable Hardware: Reconfigurable hardware allows programmers to parameterize high-level descriptions of algorithms and allocate resources based on bottleneck of the application. The volume of the design space grows exponentially with each added parameter, and most applications have over a billion configurations. It is intractable for either a human or a brute-force search engine to explore the entire space, so Spatial provides built-in automated design space exploration (DSE) (Koeplinger et al. 2018). This tool uses either a heuristic approach or HyperMapper (Nardi et al. 2019), a multi-objective optimizer, to quickly determine the fastest design points that fit in the target accelerator. Spatial contains a variety of ML models to estimate resource utilization and performance, which allows design point evaluation to bypass place and route (PaR) and complete in milliseconds.

3.1.3 Spatial to Plasticine Compiler

Plasticine's programming abstraction supports efficient, distributed data-flow execution. Unlike processors with dynamic instruction fetch and decode, Pattern Compute Units (PCUs) are statically configured to execute a small window of instructions as part of a vectorized map-reduce. The PCU consumes and produces pipelined data. Communication between compute and on-chip memory is streaming to avoid the overhead of a request- response protocol. This execution model maximizes compute throughput by using deep pipelining, instruction-level parallelism, data-level parallelism, and kernel-level parallelism, without the performance and energy overhead incurred by dynamic execution.

Nonetheless, a pure data-flow abstraction can only express very limited sets of applications. Spatial provides the ability to express nested loops, parallel patterns, branching, and FSMs, which are required to capture efficient implementations of a broader set of applications. By supporting the features available in Spatial, Plasticine can support a wide range of application domains while maintaining its high compute throughput and energy efficiency.

There are two key challenges involved in supporting arbitrary Spatial on Plasticine. First, Spatial captures the control flow graph of the program with a hierarchy of controllers to schedule program execution. Implementing this hierarchy over a flexibly- timed, distributed network requires excessive acknowledgment messages, and therefore large performance overheads, to guarantee correctness. Second, users can express arbitrarily large computations and data structures in software, which must be mapped onto fixed-size hardware blocks. This process is similar to FPGA high-level synthesis from a C-like abstraction to Verilog. However, inefficient mapping results in worse resource use and performance loss due to Plasticine's coarse configuration granularity. To address the first challenge, the compiler converts the controller hierarchy to a distributed data-flow graph and introduces dummy data dependencies to implement imperative constructs such as Do-While loops, branching, on-chip memory consistency, etc. Next, the compiler partitions the program based on a hardware specification to fit the program onto the hardware resources.

Program Synthesis and Partitioning: When mapping loop constructs onto Plasticine, the Plasticine compiler pipelines basic blocks' data-flow graphs over PCU stages and vectorizes inner loops across PCU lanes. To support pipelining, the compiler duplicates and distributes the controller hierarchy across CUs. These duplicated controllers are used to rate-match streams between CUs without additional synchronization, allowing pipelining at full throughput over the network. The compiler optimizes the allocation of the memories based on whether they are locally or globally accessed. The compiler also optimizes memories that are only accessed locally, pruning guaranteed-inactive network links to aid placement. To avoid data hazards in static random-access memory (SRAM), such as Write After Write (WAW) and Write After Read (WAR), the compiler inserts control tokens in the data- flow graph to convert control dependencies into data dependencies.

To map basic blocks larger than a PCU, the compiler partitions large basic blocks into multiple subgraphs that each can fit into a PCU, as shown in Figure 4(a). When Spatial parallelizes a program, it banks intermediate buffers to scale read and write bandwidth accordingly. Figure 4(b) shows how the compiler maps a logical SRAM that exceeds the capacity or banking limit of a single pattern memory unit (PMU) using multiple PMUs. In this case, the computation is parallelized by a factor of two. The compiler first allocates Address Generators (AGs) to scatter the vectorized requests to all PMUs, and the vectorized responses are joined for each of the two parallel compute threads by a PCU. If a basic block consumes more inputs than a PCU has input ports, the basic block is partitioned and mapped to multiple PCUs. For special cases, the compiler can analyze the access patterns of the parallel compute threads and assign banks among PMUs such that the crossbars between AGs, PMUs, and PCUs are eliminated.

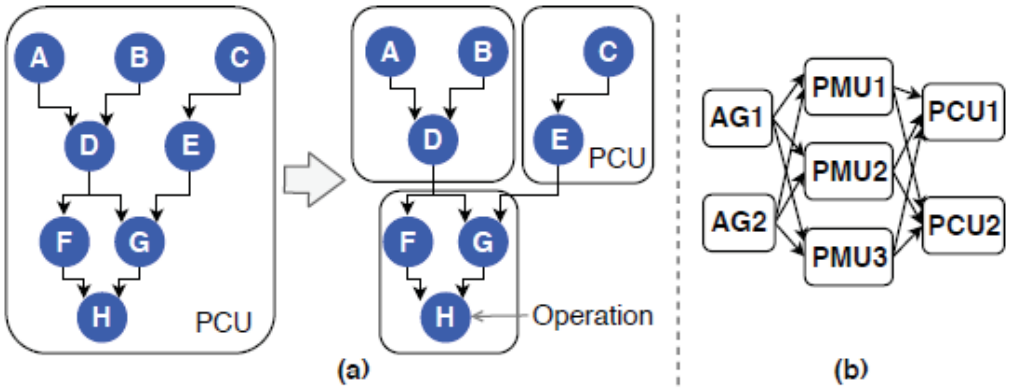

Figure 4: Partitioning a Large Data-flow Graph

Compiler-Directed Place and Route: The final pass of mapping is to PaR the final dataflow graph onto the accelerator array. The introduction of a hybrid static-dynamic network guarantees success of this process. Nonetheless, sharing dynamic network links for bandwidth-sensitive traffic decreases performance. Our recent work (Zhang et al. 2019) shows that static program information assists the PaR tool in identifying high-priority communication. With programmer-annotated input sizes, the compiler derives the expected iteration count of all loops, which is used to derive activation counts for links. Even without annotation, relative activation frequency can be inferred from the controller structure. This information is used to drive heuristics during PaR that map high-bandwidth links on the static network to guarantee bandwidth and leave low-bandwidth links on the dynamic network.

3.2 TA1: Hardware

3.2.1 Plasticine I

The original Plasticine architecture (Prabhakar, et al. 2017) uses Spatial's parallel patterns-based programming model and flexible domain-specific hardware to efficiently achieve high performance. Plasticine I is a checkerboard array of 64 PCUs and 64 PMUs as shown in Figure 5. Each PCU has 16 single instruction multiple data (SIMD) lanes and 6 SIMD stages, for a total of 96 Functional Units (FUs). Each PMU has 16 banks of 16 KiB each (256 KiB). This provides a theoretical maximum of 6 tera floating-point operations per second (TFLOPS), 16 MB on-chip storage, and 4 TBps on-chip bandwidth. The on-chip resources connect to a double data rate type 3 (DDR3) memory system with support for dense and sparse accesses and 50 GBps bandwidth.

The commercial implementation of Plasticine I by SambaNova systems keeps the basic architecture, but dramatically increases the number of PCUs and PMUs and the amount of on-chip memory and support for different floating-point data types such as BFloat16. The peak performance of the SambaNova chip is above 300 TFLOPS.

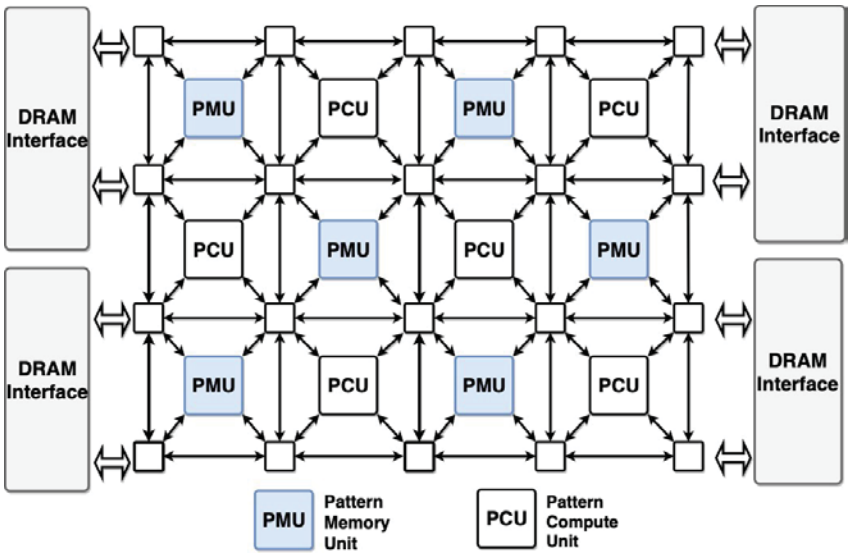

Figure 5: Plasticine I Architecture

3.2.2 Plasticine II

The goal of Plasticine II is to support converged, multi-domain data analytics applications, with a focus on avoiding the siloing that can occur when different parts of an application are sent to different accelerators. In a Plasticine II program, the user will be able to program in a variety of paradigms: TensorFlow or PyTorch for machine learning, SQL for relational data analytics, SNAP for graph algorithms, P4 for networking, and more. Our compute fabric will use the key advantages of Plasticine I – spacious memory, high compute bandwidth, and a fast interconnect – with minor changes to support this menagerie of new applications.

We optimized the architecture of Plasticine II for the provided SDH workloads and for several key emerging applications for data analytics. To support the SDH provided applications, we have conducted studies into efficient parallelization for dense ML models. We discuss our work to optimize Plasticine I's network fabric, with the goal of achieving better energy efficiency and understanding the scaling limits imposed by the network. We have also investigated integrating support for relational database operations into Plasticine II and integrating Plasticine II into an intelligent networking system.

Efficient Network on Chips (NoCs): Applications with different characteristics often have conflicting requirements for network bandwidth. Compute-intensive applications create high-throughput streaming traffic from distributed compute to on-chip scratchpads, whereas input/output (IO)-bound applications utilize few on-chip resources, including on-chip network bandwidth. Plasticine I used a static on-chip network with enough bandwidth to saturate its compute resources. However, applications that do not map into the static network will not run at all, requiring resource overprovisioning to support future, unseen applications.

Alternatively, a dynamic NoC provides flexibility with better link utilization but is area- and energy-inefficient. We show that a low-bandwidth and infrequently used dynamic network, coupled with a high-bandwidth static network, provides sufficient bandwidth for streaming applications while reducing the routing overhead in the static network (Zhang et al. 2019).

Additionally, the dynamic network provides an escape path for traffic hotspots, decreasing the overall routing distance. With less data movement, the hybrid network provides a 1.8x energy efficiency improvement and 2.8x performance improvement over the pure static and dynamic networks, respectively.

Database Acceleration: With emerging success in ML, ML has been widely adopted in many application domains. However, real-world systems often acquire and preprocess the data with the database (DB) queries before feeding data into the ML models. A key feature in Plasticine II is the ability to tackle mixed ML-DB benchmarks, which maximizes compute throughput in ML and memory bandwidth in DB.

We introduce a variety of microarchitectural features to support database queries, including vectorized filtering, join, group-by reduction, and sorting. We show that communication bandwidth between DB and ML can be the bottleneck in end-to-end application performance, which requires a general-purpose data analytics accelerator that can efficiently map both applications at the same time.

On-chip Sparsity: Support for on-chip sparsity is a key driver of our Phase 2 evaluation effort. Currently, Plasticine is able to deliver its performance due to the large amount of on-chip memory bandwidth from its user-configured scratchpads. To assist the user, Spatial banks these scratchpads for parallel accesses: up to 16 accesses per PMU, per cycle, for a total of 4 TBps of aggregate memory bandwidth to 16 MB of memory. This banking resolves both *structural hazards* (two readers cannot access the same bank in the same cycle) and *data hazards* (parallelization and pipelining leading to incorrect program execution). However, because banking relies on static analysis of accesses to memory (based on loop iterators), it cannot be applied to dynamic applications, which access memory indirectly through random pointers. We have developed two new techniques (dynamic dependency resolution and hardware-assisted memory locking) to resolve these hazards and maintain bandwidth for sparse applications.

Plasticine for Networking: Plasticine has low latency due to its fine-grained pipelined computation; furthermore, Plasticine's latency is deterministic because there are no background tasks to delay processing. We explored specializing Plasticine's architecture for inference, with minimal hardware overhead, by shrinking the size of on-chip memories, eliminating DRAM controllers, and reduced precision (Swamy et al. 2019). This reduced architecture could function inside a high-performance network switch as shown in Figure 6 and evaluate ML models on each packet with minimal added latency, supporting applications like anomaly detection.

Figure 6: Plasticine in a Networking Switch for Ultra-low-latency Inference

4. RESULTS AND DISCUSSION

4.1 SambaNova Systems Plasticine I Implementation

SambaNova Systems has developed an improved version of the Plasticine I architecture called the Cardinal SN10. The SN10 chip which is shown in Figure 7 is implemented in 7nm TSMC process technology and contains 40 billion transistors and 50 Km of wire. Cardinal is a Reconfigurable Dataflow Unit (RDU) architecture based on Plasticine I. The SN10 chip contains 640 PMUs and 640 PCUs, and a total 320MB of on-chip SRAM in the PMUs. The SN10 can be connected to 1.5 TB of external DRAM. Multi SN10s can be connected together using RDU-connect which is a direct high-performance communication channel between RDUs. Figure 8 shows the first SambaNova system that will be available; it is called the SN10-8 and contains 8 Cardinal SN10 chips and has a peak floating-point performance of 2.6 petaflops. These RDU based systems are already showing substantial improvements in performance and energy efficiency compared to graphics processing unit (GPU) based systems on very large ML training problems.

Figure 7: SambaNova Cardinal SN10 Chip

Figure 8: SambaNova SN10-8 System with 12 TB DRAM in ¼ Rack

4.2 ML Models with Complex Dataflow on Plasticine

Most state-of-the-art ML models contain kernels with complex data-flow graphs. During Phase 1, we evaluated Plasticine's performance on one popular class of these kernels, serving Recurrent Neural Networks (RNNs) in real time (Zhao et al. 2019).

RNN applications are an important class of artificial intelligence (AI)-powered, low-latency data center workloads. Around 30% of ML workloads at Google are RNNs. Despite RNNs' popularity, it is hard for existing ML accelerators to accelerate RNNs efficiently.

The two major challenges are: first, existing ML accelerators assume that matrix multiplication dominates the computation in applications. However, inference services, such as real-time translation, cannot batch requests, which only contains matrix-vector multiplications and element-wise multiplications. These operations have a smaller compute-to-data ratio than matrix multiplication. Hence, most ML accelerators suffer from poor hardware utilization when serving RNNs. Second, RNN kernels consist of many separate operations: existing ML accelerators must frequently buffer intermediate data, leading to low energy efficiency.

Based on these observations, we propose that an accelerator supporting a fine-grained abstraction would provide better performance and higher efficiency. Specifically, an accelerator must support pipelining arbitrary loop nests to achieve high efficiency when serving ML models with complex dataflow. With this support, cross-kernel fusion reduces memory footprints and enhances hardware utilization.

We implemented the RNN kernels from the DeepBench benchmark in Spatial and evaluated performance with Plasticine at 28nm. We compared our results to Microsoft's BrainWave architecture on a Stratix 10 FPGA, and NVIDIA's V100 GPU, at 14nm and 12nm technologies, respectively. Overall, we found that Plasticine provides a geometric mean speedup of 30x compared to the Tesla V100 GPU, and 2x compared with Microsoft's BrainWave due to better utilization of the compute FLOPS.

4.3 Database Acceleration

The preliminary performance increase relative to an auto-parallelized Apache MADLib system, running on Postgres, is shown in Figure 9. A variety of mixed DB-ML training and inference operations running on Plasticine have a geometric mean speedup of 2376x.

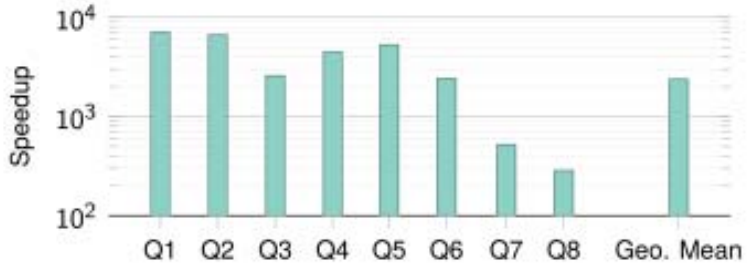

Figure 9: The Speedup of our Plasticine Database Solution over Apache MADLib

High-Bandwidth Memory (HBM) Scratchpad: Many database operations, like sorting, require materializing intermediate values to scratchpads. Plasticine I had a memory system with two levels of hierarchy: extremely fast, but small, on-chip SRAM and slow DDR3 DRAM. As part of our database project, we are exploring a three-level memory hierarchy: SRAM, HBM, and DDR4 DRAM; this will allow tiled applications to run faster.

Microarchitectural Advancements: As shown in Figure 10, we added support to Plasticine II's design for record data types. We extend the PCU's logic to flexibly process these datatypes, based around a variable-length dequeue buffer and vectorized comparisons. For example, we can join two tables by comparing one's join key against every lane in parallel, or we can group elements by counting adjacent keys.

Sorting is more complicated and is performed by reversing one input vector and comparing, elementwise, against the other. This determines the number of elements to take from each list, which are then themselves sorted using the PCU fabric as a sorting network.

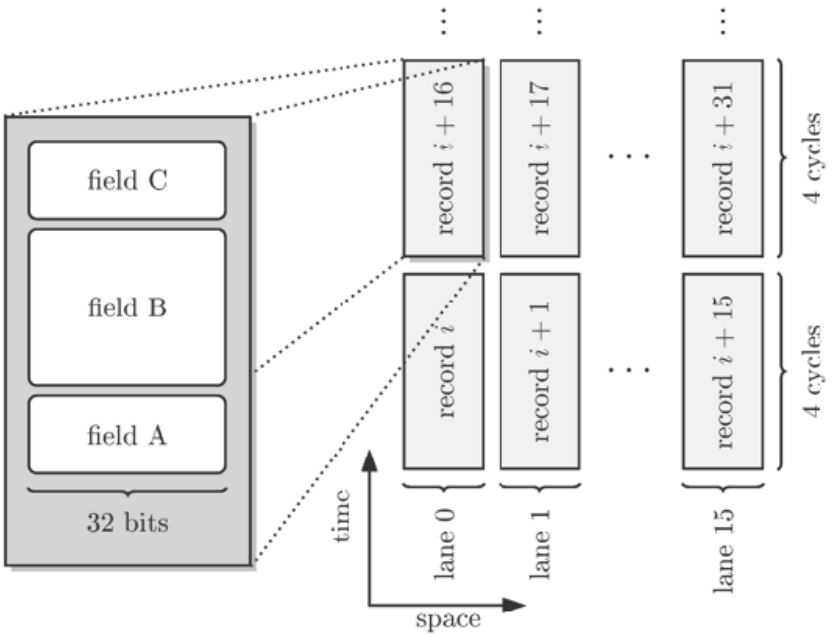

Figure 10: Record Data Streams through Plasticine II's Fabric in Time
Each record remains in one lane, with fields moving through the lane over multiple cycles.

The key used for any decision is always reordered to be the first element in the record. After a decision is made (e.g., which elements to take for a sort operation), it is tracked for the remaining vector elements, keeping them together as they proceed through the pipeline. To enable performing operations on multiple keys, the input logic supports reordering fields in a record by dequeuing the input first-in, first-out (FIFO) out of order. Throughout the database logic, provisions are made for partial lists using valid and done signals: it is not possible to determine beforehand the number of elements in a list, so the hardware must dynamically adapt.

4.4 Discussion

The main goal of the SambaNova Plasticine I implementation is to demonstrate the real benefits of SDH in the form of reconfigurable dataflow using an industrial strength design team. The result of this exercise is a very-high performance SN10 RDU chip and systems based on this chip. This main goal has been achieved as SN10 based systems have demonstrated far superior performance and energy efficiency compared to GPUs and FPGAs on machine learning applications. The secondary goal of developing a real Plasticine implementation is to serve as a software development platform for future Plasticine compiler and application software to be developed in in Phase 2 and Phase 3 of the SDH program. Since we were not awarded Phase 2 funding this secondary goal will not been realized.

5. CONCLUSIONS

The overall goal of the Plasticine SDH project is to develop a universal data analytics accelerator with support for dynamic reconfiguration based on domain specific languages compilation technology and the parallel-pattern based Plasticine reconfigurable dataflow architecture. We have developed a commercial implementation of the Plasticine architecture called the SambaNova Systems Cardinal SN10. This real system is already demonstrating the dramatic performance and energy efficiency gains possible with reconfigurable dataflow. We have also investigated enhancements to the base Plasticine architecture to support irregular data access and irregular dataflow to enable sparse matrix applications, graph analytic applications and database applications to run efficiently. These architecture improvements are required to make Plasticine truly universal for data analytics. We have developed a complete software and compiler stack for Plasticine. At the end of Phase 1 this stack can take TensorFlow machine learning programs and translate them into optimized Plasticine configurations using Spatial a new performance oriented (hardware) DSL and sophisticated compiler optimization algorithms.

6. PUBLICATIONS SUPPORTED BY SDH

1. Zhao, T., Zhang, Y. and Olukotun, K., "Serving Recurrent Neural Networks Efficiently with a Spatial Accelerator". In *Proceedings of the 2 nd SysML Conference*, Palo Alto, CA, USA, 2019.
2. Zhang, Y., Rucker, A., Villim, M., Prabhakar, R. and Olukotun, K., "Scalable Interconnects for Reconfigurable Spatial Architectures". In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019.
3. Swamy, T., Rucker, A., Shahbaz, M., Yadwadkar, N., Zhang, Y. and Olukotun, K., "Taurus: An Intelligent Data Plane". In *P4 Workshop*, 2019.
4. Hadjis, S. and Olukotun, K., "TensorFlow to Cloud FPGAs: Tradeoffs for Accelerating Deep Neural Networks". In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 360-366, Sep. 2019.
5. Rucker, A., Shahbaz, M., Swamy, T. and Olukotun, K., "Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs". In *Proceedings of the 3rd Asia- Pacific Workshop on Networking 2019*, pp. 71–77, Beijing, China, 2019.
6. Singhal, R., Zhang, Y., Ullman, D., Prabhakar, R. and Olukotun, K., "Efficient Multiway Hash Join on Reconfigurable Hardware". In *Technology Conference on Performance Evaluation and Benchmarking*, Los Angeles, CA, USA, 2019.
7. Koeplinger, D., Feldman, M., Prabhakar, R., Zhang, Y., Hadjis, S., Fiszel, R., Zhao, T., Nardi, L., Pedram, A., Kozyrakis, C. and Olukotun, K., "Spatial: A Language and Compiler for Application Accelerators". In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 296--311, Philadelphia, PA, USA, 2018.
8. Nardi, L., Koeplinger, D. and Olukotun, K., "Practical Design Space Exploration". In *CoRR*, Vol. abs/1810.05236, 2018.
9. Prabhakar, R., Zhang, Y., Koeplinger, D., Feldman, M., Zhao, T., Hadjis, S., Pedram, A., Kozyrakis, C. and Olukotun, K., "Plasticine: A reconfigurable architecture for parallel patterns". In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 389-402, June 2017.
10. De Sa, C., Feldman, M., Ré, C. and Olukotun, K., "Understanding and optimizing asynchronous low-precision stochastic gradient descent". In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 561-574, June 2017.
11. Koeplinger, D., Prabhakar, R., Zhang, Y., Delimitrou, C., Kozyrakis, C. and Olukotun, K., "Automatic Generation of Efficient Accelerators for Reconfigurable Hardware". In *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 115--127, Seoul, Republic of Korea, 2016.
12. De Sa, C., Olukotun, K. and Re, C., "Ensuring Rapid Mixing and Low Bias for Asynchronous Gibbs Sampling". In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, pp. 1567--1576, New York, NY, USA, 2016.

APPENDIX: PLASTICINE II PERFORMANCE REPORT

1. Background

We are building an efficient and universal data analytics accelerator system by leveraging our research into DSLs and parallel patterns-based hardware. The Spatial [1] language is a hardware-centric DSL for reconfigurable architectures and is based around a parallel patterns abstraction. The original Plasticine architecture [2] uses Spatial's parallel patterns-based programming model, with flexible domain-specific hard-ware to efficiently achieve high performance. We showed that coarse-grained reconfigurable accelerators provide a promising abstraction for data analytics, as they avoid the overheads associated with instruction-based processors, such as instruction fetch and dynamic operation reordering. Plasticine II and Constellation, a new front-end for Spatial, will use additional parallel patterns to support a wider range of applications.

In this report, we present our accomplishments in Phase 1 of Plasticine II development and the projected features of Plasticine II in Phase 2 and beyond. We will enable users to program in multiple paradigms, including databases, dense neural networks, sparse applications, and graph analytics, without sacrificing efficiency. To improve developer productivity, we introduce a new compiler stack that maps programs written in high-level DSLs to Plasticine II, with Spatial as an intermediate representation. To improve efficiency, we extend the original Spatial with new optimizations for more general parallel patterns shared across multiple domains, including new patterns like filter, and flatMap. This more general set of parallel patterns further drives the architectural design of Plasticine II. We detail these extensions in Sections 2 and 3. Finally, we show an evaluation of Phase 1 Plasticine II with estimated features from Phase 2 to handle applications from multiple domains.

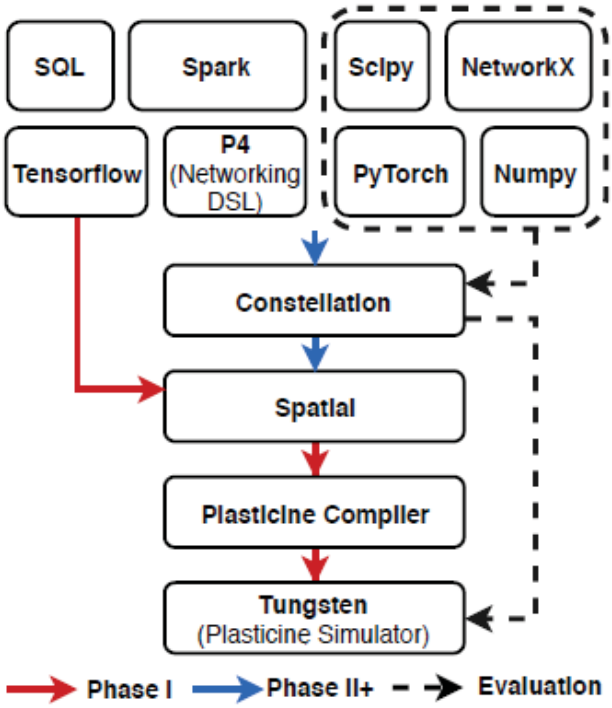

Figure 1: An Overview of the Complier Stack

2. TA2: The Software and the Compiler Stack

2.1 Phase 1

In this section, we first describe the existing compiler at the end of Phase 1 with various optimization performed at different levels of the stack. Finally, we present a case study on the performance of Plasticine compared to other accelerator frame-works.

2.1.1 TensorFlow to Spatial Compiler

To improve developer productivity, we developed an end-to-end, open-source compiler that maps TensorFlow to Spatial [3]. The compiler is capable of running state-of-the-art DNNs specified in TensorFlow for applications ranging from object recognition to speech-to-text translation.

The compiler flow is summarized in Figure 2.

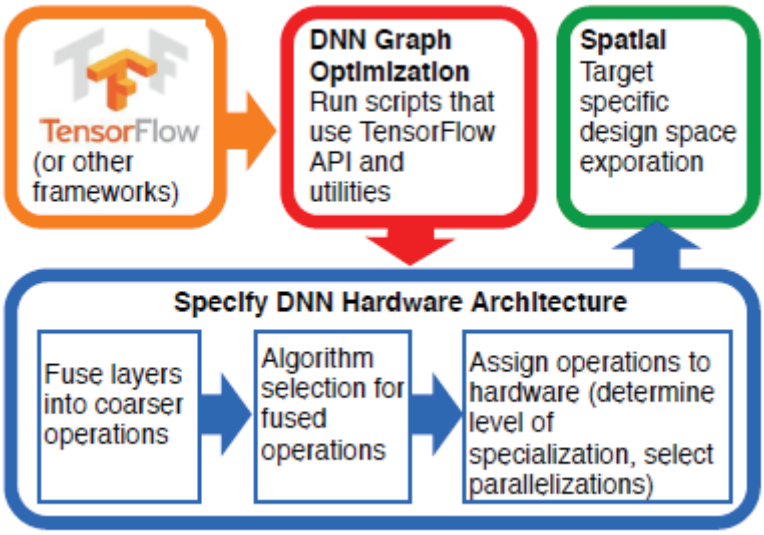

Figure 2: TensorFlow to Spatial Compiler

The compiler leverages both TensorFlow and Spatial to perform optimizations at multiple abstraction levels. In addition to allowing developers to synthesize accelerators from a high-level frame-work, the compiler is also a hardware architecture exploration tool. Spatial's flexibility allows experimentation with various architecture choices [3]. This allows developers to explore design space decisions, including accelerator operation granularity, hardware specialization, algorithm transformation, tensor storage format, and memory-level parallelism. To do this, the compiler relies on Spatial's ability to synthesize efficient hardware for many types of circuits, discussed in detail in the next section.

2.1.2 Spatial

Spatial is a hardware-centric DSL that expresses applications using parallel patterns. Unlike most software languages, Spatial has an explicit memory hierarchy, including syntax for transfers (dense and sparse) between DRAM and on-chip memories. To speed up programs, Spatial allows the programmer to parallelize loops and employ scheduling directives dictating how loops execute relative to one another. It also automatically re-times pipelined operations and detects common reduction patterns that can be optimized with specialized hardware. To scale on-chip memory band-width, Spatial analyzes memory access patterns and uses the results to bank and buffer on-chip memories.

Accumulation Optimizations Spatial can detect a variety of accumulation patterns referencing non-addressable memories (i.e., `Registers`) and transform the IR to optimize these cases. However, accumulations into addressable memories could require the compiler to throttle performance for guaranteed correctness. Consider the accumulation `sram(f(i)) = g(sram(h(i)))` inside a loop body. With the loop-carried dependency, the trivial solution to guarantee correctness is to stall each iteration until the previous iteration has up-dated the memory. However, there are many cases where this is too conservative, including low-rank accumulations and regional accumulations. In-stead, Spatial pipelines different iterations of the loop by analyzing the relationship between `f(i)` and `h(i)`, the range of iterator `i`, and the complexity of function `g`. Spatial contains an analysis pass for accumulations into addressable memory, determining the iteration space and computing the minimum number of iterations the controller must wait before it can issue the next iteration (i.e., the initialization interval).

Banking Optimizations For each on-chip memory, Spatial analyzes all accesses and uses the controller hierarchy to identify which accesses must be time-multiplexed and which are bankable. If the concurrent accesses are derived from loop un-rolling, the compiler determines a banking assignment for each unrolled lane as an affine function of the unrolled loop iterators. Next, the compiler tests the validity of the memory partitioning using conflict polytope emptiness testing [4]. At run-time, accesses from different groups are active at different times. Each concurrent access within the group carries a bank ID and an offset within each bank that gets dynamically resolved. Figure 3 shows two groups of time-multiplexed accesses, each with four parallel accesses. In cases when the banking IDs (b0-b7) can be statically resolved, the compiler eliminates the crossbar.

Spatial handles a variety of access patterns that can contain both affine and non-affine components. It can also bank for concurrent accesses, as long as it detects synchronization guarantees on the iterators. One example of each case is described below:

- `sram(a1*i+c1+lut(i))`, for constant `a1`, unrolled iterator `i`, loop-invariant value `c1`, and loop-variant value `lut(i)`, is bankable because of the affine component `a1*i+c1`, but requires a crossbar because the non-affine component `lut(i)` cannot be statically resolved.
- `sram(2*i-1)` and `sram(2*i)` can be banked as long as unrolled values of iterator `i` always increment during the same cycle.

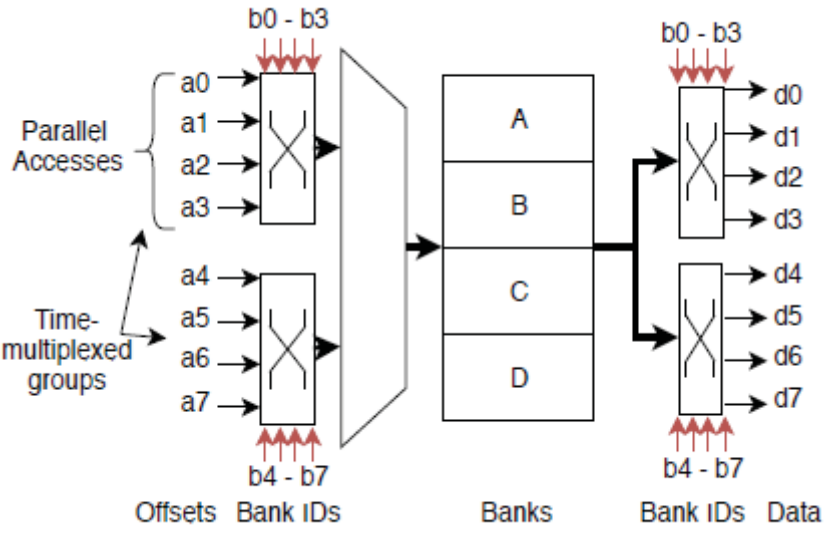

Figure 3: On-chip Memory Banking

Banking Cost Models Although many memories can be parallelized using a variety of banking schemes, some schemes may be more hardware-efficient than others. Spatial can compute multiple candidate schemes and choose the best one. By using an ensemble of decision-tree based models and deep learning models, it computes the cost of both the hardware used to realize the banking scheme and the auxiliary nodes inserted into the address path to handle bank resolution. This provides a more accurate estimate of the cost than heuristics like bank switching [5] and partition volumes.

Banking Report and Annotations The compiler generates a report showing each memory's valid banking schemes and their estimated cost; the user can sort by estimated cost or compiler search time. Understanding which accesses consume the most hardware resources allows the user to refactor programs to use less expensive banking schemes. Similarly, knowing which memories the compiler struggles to bank allows the programmer to add annotations that guide the compiler toward the correct banking scheme more quickly. Because Spatial spends most of its time analyzing banking, addressing these issues decreases compilation time significantly.

Do-While Solution Many sparse and graph applications employ dynamic, data-dependent control flow patterns, like "run until converged" and "process an unknown number of incoming packets." Although these map well to procedural soft-ware, they are tricky to implement in hardware. Spatial has Do-While loops that capture these situations; these loops are implemented using infinite counter chains and explicit break conditions. Spatial also allows users to implement arbitrary Finite State Machines (FSMs) to capture non-linear iteration spaces.

Design Space Exploration for Reconfigurable Hardware Reconfigurable hardware allows programmers to parameterize high-level descriptions of algorithms and allocate resources based on bottleneck of the application. The volume of the de-sign space grows exponentially with each added parameter, and most applications have over a billion configurations. It is intractable for either a human or a brute-force search engine to explore the entire space, so Spatial provides

built-in automated design space exploration (DSE) [6]. This tool uses either a heuristic approach or HyperMapper [7, 8], a multi-objective optimizer, to quickly determine the fastest design points that fit in the target accelerator. Spatial contains a variety of ML models to estimate resource utilization and performance, which allows design point evaluation to bypass place and route (PaR) and complete in milliseconds.

2.1.3 Spatial to Plasticine Compiler

Plasticine's programming abstraction supports efficient, distributed data-flow execution. Unlike processors with dynamic instruction fetch and decode, PCUs are statically configured to execute a small window of instructions as part of a vectorized map-reduce. The PCU consumes and produces pipelined data. Communication between compute and on-chip memory is streaming to avoid the overhead of a request-response protocol. This execution model maximizes compute throughput by using deep pipelining, instruction-level parallelism, data-level parallelism, and kernel-level parallelism, without the performance and energy overhead incurred by dynamic execution.

Nonetheless, a pure data-flow abstraction can only express very limited sets of applications. Spatial provides the ability to express nested loops, parallel patterns, branching, and FSMs, which are required to capture efficient implementations of broader applications. By supporting the features available in Spatial, Plasticine can support a wide range of application domains while maintaining its high compute throughput and energy efficiency.

There are two key challenges involved in sup-porting arbitrary Spatial on Plasticine. First, Spatial captures the control flow graph of the pro-gram with a hierarchy of controllers to schedule program execution. Implementing this hierarchy over a flexibly-timed, distributed network re-quires excessive acknowledgment messages, and therefore large performance overheads, to guarantee correctness. Second, users can express arbitrarily large computations and data structures in software, which must be mapped onto fixed-size hardware blocks. This process is similar to FPGA high-level synthesis from a C-like abstraction to Verilog. However, inefficient mapping results in worse resource use and performance loss due to Plasticine's coarse configuration granularity. To address the first challenge, the compiler converts the controller hierarchy to a distributed data-flow graph and introduces dummy data a dependencies to implement imperative constructs such as Do-While loops, branching, on-chip memory consistency, etc. Next, the compiler partitions the program based on a hardware specification to fit the program onto the hardware resources.

Program Synthesis and Partitioning When mapping loop constructs onto Plasticine, the Plasticine compiler pipelines basic blocks' data-flow graphs over PCU stages and vectorizes inner loops across PCU lanes. To support pipelining, the compiler duplicates and distributes the controller hierarchy across CUs. These duplicated controllers are used to rate-match streams between CUs without additional synchronization, allowing pipelining at full throughput over the network. The compiler also optimizes memories that are only accessed locally, pruning guaranteed-inactive network links to aid placement. To avoid data hazards in SRAM, such as Write After Write (WAW) and Write After Read (WAR), the compiler inserts control tokens in the data-flow graph to convert control dependencies into data dependencies.

To map basic blocks larger than a PCU, the compiler partitions large basic blocks into multiple subgraphs that each can fit into a PCU, as shown in Figure 4(a). When Spatial parallelizes a program, it banks intermediate buffers to scale read and write bandwidth accordingly. Figure 4(b) shows how the compiler maps a logical SRAM that exceeds the capacity or banking limit of a single PMU using multiple PMUs. In this case, the computation is parallelized by a factor of two. The compiler first allocates Address Generators (AGs) to scatter the vectorized requests to all PMUs, and the vectorized responses are joined for each of the two parallel compute threads by a PCU. If a basic block consumes more inputs than a PCU has input ports, the basic block is partitioned and mapped to multiple PCUs. For special cases, the compiler can analyze the access patterns of the parallel compute threads and assign banks among PMUs such that the crossbars between AGs, PMUs, and PCUs are eliminated.

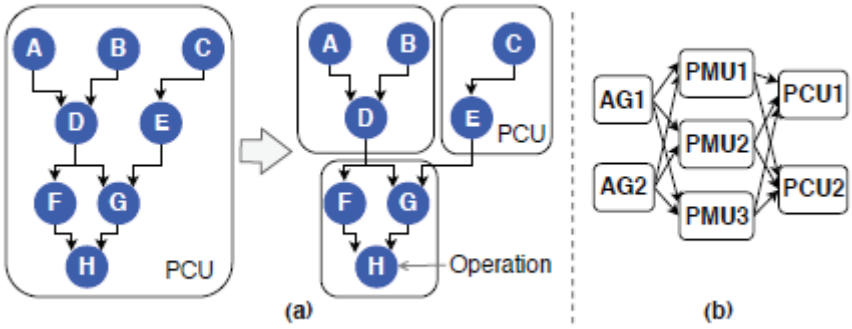

Figure 4: Partitioning a large Data-flow Graph

Compiler-Directed Place and Route The final pass of mapping is to PaR the final dataflow graph onto the accelerator array. The introduction of a hybrid static-dynamic network guarantees success of this process. Nonetheless, sharing dynamic network links for bandwidth-sensitive traffic de-creases performance. Our recent work [9] shows that static program information assists the PaR tool in identifying high-priority communication. With programmer-annotated input sizes, the compiler derives the expected iteration count of all loops, which is used to derive activation counts for links. Even without annotation, relative activation frequency can be inferred from the controller structure. This information is used to drive heuristics during PaR that map high-bandwidth links on the static network to guarantee bandwidth and leave low-bandwidth links on the dynamic network.

2.1.4 Case Study: ML Models with Complex Dataflow on Plasticine

Most state-of-the-art ML models contain kernels with complex data-flow graphs. During Phase 1, we evaluated Plasticine's performance on one popular class of these kernels, serving Recurrent Neural Networks (RNN) in real time [10]. RNN applications are an important class of AI-powered, low-latency data center workloads. According to [11], around 30% of ML workloads at Google are RNNs. Despite RNNs' popularity, it is hard for existing ML accelerators [11, 12, 13, 14, 15] to accelerate RNNs efficiently.

The two major challenges are: first, existing ML accelerators assume that matrix multiplication dominates the computation in applications. How-ever, inference services, such as real-time translation, cannot batch requests, which only contains matrix-vector multiplications and element-wise multiplications. These operations have a smaller compute-to-data ratio than matrix multi-plication. Hence, most ML accelerators suffer from poor hardware utilization when serving RNNs. Second, RNN kernels consist of many separate operations: existing ML accelerators must frequently buffer intermediate data, leading to low energy efficiency.

Based on these observations, we propose that an accelerator supporting a fine-grained abstraction would provide better performance and higher efficiency. Specifically, an accelerator must support pipelining arbitrary loop nests to achieve high efficiency when serving ML models with complex dataflow. With this support, cross-kernel fusion reduces memory footprints and enhances hardware utilization.

We implemented the RNN kernels from the DeepBench [16] benchmark in Spatial and evaluated performance with Plasticine at 28nm. We compared our results to Microsoft's BrainWave architecture on a Stratix 10 FPGA, and NVIDIA's V100 GPU, at 14nm and 12nm technologies, respectively. Overall, we found that Plasticine pro-vides a geometric mean speedup of 30x compared to the Tesla V100 GPU, and 2x compared with Microsoft's BrainWave due to better utilization of the compute FLOPS.

2.2 Phase 2 and Beyond

During Phase 2 we will focus on addressing two major challenges:

1. Many emerging domains contain applications that cannot be easily decomposed into the parallel patterns defined in Phase 1, but we want to provide users with high performance without sacrificing the expressibility of their programming tools.
2. Rapid reconfiguration allows accelerators to support dynamic applications with changing data patterns. However, such rapid reconfigurability requires that the high-level compilers perform Just In Time (JIT) PaR.

For challenge 1, we will enrich the key pat-terns from Phase 1 with new patterns from various emerging domains, listed in Table 1. To better sup-port the sparse accesses needed for graph applications and sparse linear algebra, we will introduce new microarchitectural features, detailed in Section 3.2.1 and 3.2.2. We will also build a compiler infrastructure that can directly process a high-level computation graph without requiring the user to significantly rewrite the code. To solve challenge 2, we will add support for flexible DSE and recompilation using captured performance counter data. We will continue to optimize the Spatial compiler to support a broader intermediate representation and more optimization passes.

Table 1. Application Domains

Domains	DSL	Key Patterns
Linear Algebra	Spark [17], OptiML [18]	`Map, Reduce, Foreach`
Sparse Linear Algebra	scipy [19]	`Map, Reduce, Scatter, Gather, Foreach`
Deep Learning	Tensorflow [20], PyTorch [21]	`Map, Reduce, Foreach`
Classic ML	Spark	`Map, Reduce, Foreach`
Database	SQL	`Merge, Join, GroupBy, Filter`
Graph	NetworkX [22], SNAP [23]	`GroupBy, Scatter, Gather` `Filter, Scan, FlatMap`

Table 2. Plasticine II's Theoretical Peak Compute Performance

Precision	fp32	fp16	fp8
per-PCU (GOPS)	96.0	192.0	384.0
Overall (TOPS)	24.6	49.2	98.3

High-Level Compiler Toolchain To support high-level applications, a new compiler infrastructure and analysis toolkit are needed. In Phase 2, we will further develop the Constellation compiler infrastructure to encode high-level, domain-specific and cross-domain optimizations and emit the necessary instrumentation for Spatial optimizations. Additionally, Constellation provides performance prediction tools to aid the low-level programs and architectural tuning. Constellation will also be able to perform JIT compilation if a user program's compute graph cannot be statically extracted.

Constellation will be built around a flexible graph-transformation framework. Constellation will use Google's MLIR infrastructure, which al-ready integrates with several existing high-level DSL, including Tensorflow. Then, a sequence of DSL-agnostic optimizations are applied; these are stored as rewrite rules from one subgraph to another in a global database and used to optimize all DSLs. A cost model based on profiled application characteristics guides the application of these subgraph-rewriting rules; after the program has been fully optimized, Spatial code is emitted. Spatial then completes additional low-level optimizations, including banking, buffering, and loop un-rolling.

ML-Guided Compilation & Synthesis One key compiler pass for reconfigurable architecture is efficiently mapping compute resources to hardware while minimizing congestion and energy: Place and Route (PaR). Most current PaR techniques, like graph-coloring allocation and simulated annealing, do not scale well and take up to several hours for a given design. We believe that ML can be used to model the behavior of PaR tools, providing information about layout optimality to higher-level compilers. However, most PaR tools are rule-based black boxes and hard to model with simple heuristics. Moreover, PaR tools' output is noisy due to the use of stochastic algorithms. No existing ML techniques for compiler optimizations perform data de-noising, which is needed to de-liver robust prediction results when modeling the PaR process.

We determined that the missing piece for modeling PaR is the absence of abundant and clean training data. To solve these, we plan to de-noise and augment the data from PaR tools by phrasing PaR prediction as a weakly-supervised regression problem [24]. We will use the augmented dataset to train light-weight ML models to predict the PaR behaviors. These models will be embedded into our new compiler stack to guide resource allocation for Plasticine II.

3. TA1: Hardware

Plasticine I The original Plasticine architecture [2] uses Spatial's parallel-patterns based programming model and flexible domain-specific hardware to efficiently achieve high performance. Plasticine I is a checkerboard array of 64 Pattern Compute Units (PCUs) and 64 Pattern Memory Units (PMUs) as shown in Figure 6. Each PCU has 16 SIMD lanes and 6 SIMD stages, for a total of 96 Functional Units (FUs). Each PMU has 16 banks of 16 kiB each (256 kiB). This provides a theoretical maximum of 6 TFLOPS, 16 MB on-chip storage, and a 4 TBps on-chip bandwidth. The on-chip resources connect to a DDR3 memory system with support for dense and sparse accesses and 50 GBps bandwidth.

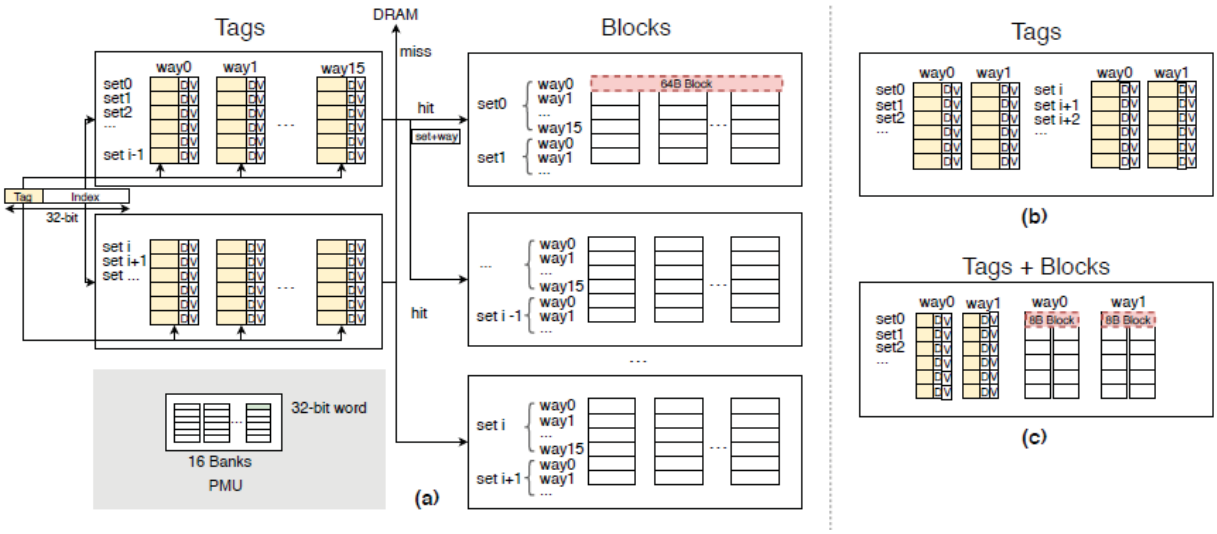

Figure 5: Composable Cache using On-chip Scratchpads in PMUs

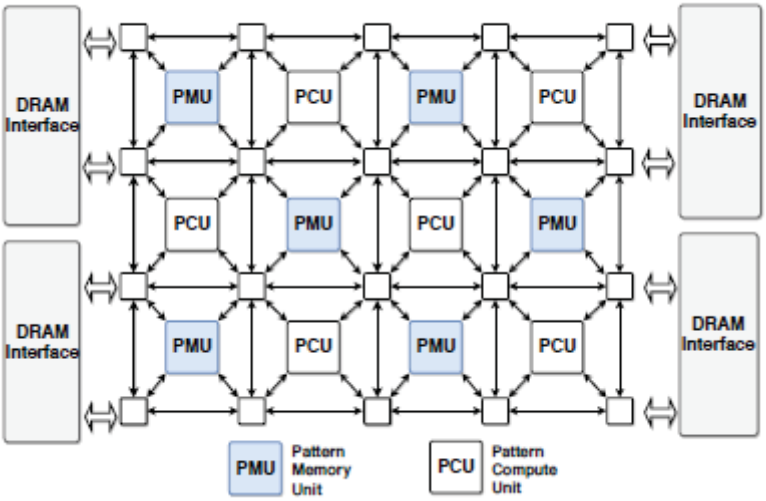

Figure 6: Plasticine Architecture

The goal of Plasticine II is to support converged, multi-domain data analytics applications, with a focus on avoiding the siloing that can occur when different parts of an application are sent to different accelerators. In a Plasticine II program, the user will be able to program in a variety of paradigms: Tensorflow or PyTorch for machine learning, SQL for relational data analytics, SNAP for graph algorithms, P4 for networking, and more. Our compute fabric will use the key advantages of Plasticine I – spacious memory, high compute bandwidth, and a fast interconnect – with minor changes to support this menagerie of new applications.

3.1 Phase 1

In Phase 1 of Plasticine II development, we explored specializations of Plasticine's compute fabric for a variety of applications: several from the provided SDH workloads and several that we believe are key emerging applications for data analytics. To support the SDH provided applications, we have conducted studies into efficient parallelization for dense Machine Learning (ML) models, including the use of subword SIMD operations for fast inference; this work is detailed in Section 2.1.4. In Section 3.1.2, we discuss our work to optimize Plasticine's network fabric, with the goal of achieving better energy efficiency and understanding the scaling limits imposed by the network. Research directions include integrating support for relational database operations (Section 3.1.3) into Plasticine II and integrating Plasticine II into a networking context (Section 3.1.4).

3.1.1 Subword SIMD Parallelism

Prior works [14, 11] have shown that low-precision inference can deliver promising performance improvements without sacrificing accuracy. Specifically, low-precision inference not only increases compute density, but also weights and intermediate data size. In our recent work [10], we explored supporting subword precision, such as `fp8` and `fp16`, without compromising the reconfigurability of Plasticine. Figure 7(a) shows a version of Plasticine's SIMD pipeline with four lanes and five stages. We introduce two low-precision struct types in Spatial: tuples of

4 8-bit (`4xfp8`) and 2 16-bit (`2xfp16`) floating-point numbers. Both types pack multiple low-precision values into single-precision storage.

Figure 7(b) shows the new opcodes for element-wise and reduction operations between the sub-word values, which reuse hardware in the single-precision operations. This microarchitectual change introduces a small hardware overhead and is local to the PCU – the on-chip memory is still accessed at 32-bit granularity. We further optimize the reduction network into the folded structure shown in Figure 7(c), which maintains the compute throughput and latency of the original reduction network while decreasing resource underutilization.

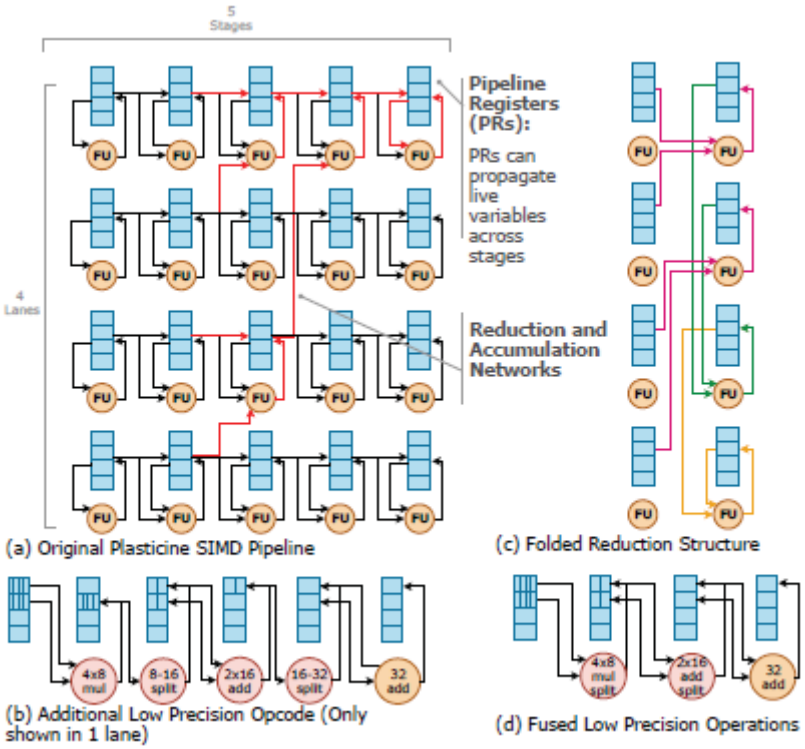

Figure 7: Subword SIMD support in Plasticine II

3.1.2 Efficient NoCs

Applications with different characteristics of-ten have conflicting requirements for network bandwidth. Compute-intensive applications create high-throughput streaming traffic from distributed compute to on-chip scratchpads, whereas IO-bound applications utilize few on-chip resources, including on-chip network bandwidth. Plasticine I used a static on-chip network with enough bandwidth to saturate its compute re-sources. However, applications that do not map into the static network will not run at all, requiring resource overprovisioning to support future, unseen applications.

Alternatively, a dynamic Network on Chip (NoC) provides flexibility with better link utilization, but is area- and energy-inefficient. In [9], we show that a low-bandwidth and infrequently used dynamic network, coupled with a high-bandwidth static network, provides sufficient bandwidth

for streaming applications while reducing the routing overhead in the static network. Additionally, the dynamic network provides an escape path for traffic hotspots, decreasing the overall routing distance. With less data movement, the hybrid net-work provides a 1.8x energy efficiency improvement and 2.8x performance improvement over the pure static and dynamic networks, respectively.

3.1.3 Database Advances

With emerging success in machine learning (ML), ML has been widely adopted in many application domains. However, real-world systems often ac-quire and preprocess the data with the database (DB) queries before feeding data into the ML models. A key feature in Plasticine II is the ability to tackle mixed ML-DB benchmarks, which maximizes compute throughput in ML and memory bandwidth in DB. We introduce a variety of microarchitectural features to support database queries, including vectorized filtering, join, group-by reduction, and sorting [25]. We show that communication bandwidth between DB and ML can be the bottleneck in end-to-end application performance, which requires a general-purpose data analytics accelerator that can efficiently map both applications at the same time. The preliminary performance increase relative to an auto-parallelized Apache MADLib system, running on Postgres, is shown in Figure 9. A variety of mixed DB-ML training and inference operations running on Plasticine have a geometric mean speedup of 2376x.

HBM Scratchpad Many database operations, like sorting, require materializing intermediate values to scratchpads. Plasticine I had a memory system with two levels of hierarchy: extremely fast, but small, on-chip SRAM and slow DDR3 DRAM. As part of our database project, we are exploring a three-level memory hierarchy: SRAM, HBM, and DDR4 DRAM; this will allow tiled applications to run faster. A summary of our proposed new memory system is shown in Table 3.

Table 3. Plasticine II's Memory System

Name	Size (MiB)	Bandwidth Dense (GBps)	Bandwidth Sparse (GBps)
SRAM	64	16 384	16 384
HBM	16 384	900	64
DDR4	1 048 576	100	12

Microarchitectural Advancements As shown in Figure 8, we added support to Plasticine II's de-sign for record data types. We extend the PCU's logic to flexibly process these datatypes, based around a variable-length dequeue buffer and vectorized comparisons. For example, we can join two tables by comparing one's join key against every lane in parallel, or we can group elements by counting adjacent keys. Sorting is more complicated, and is performed by reversing one input vector and comparing, elementwise, against the other. This determines the number of elements to take from each list, which are then themselves sorted using the PCU fabric as a sorting network.

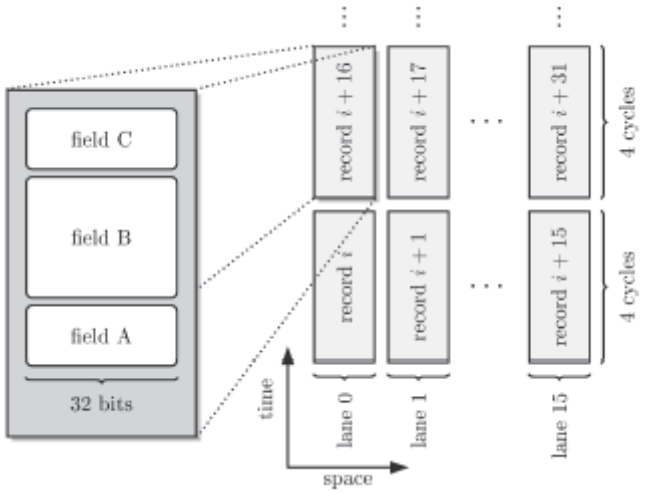

Figure 8: Record Data Streams through Plasticine II's Fabric in Time
Each record remains in one lane, with fields moving through the lane over multiple cycles

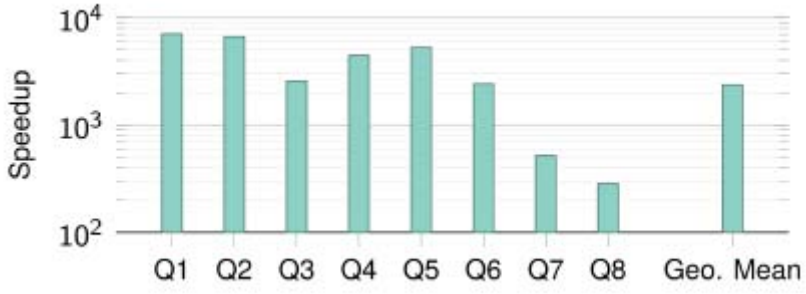

Figure 9: The Speedup of our Plasticine Database Solution over Apache MADLib

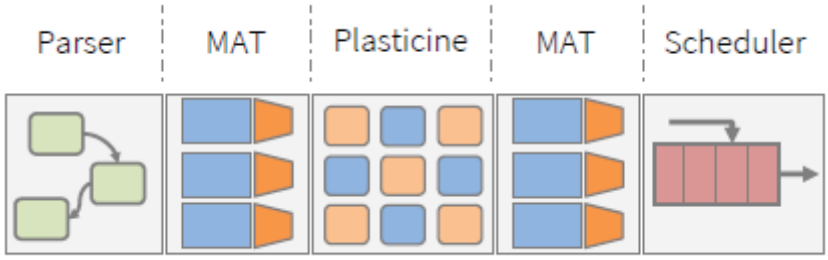

Figure 10: One Proposal to Integrate Plasticine into a networking switch for ultra-low-latency inference.

The key used for any decision is always re-ordered to be the first element in the record. After a decisions is made (e.g., which elements to take for a sort operation), it is tracked for the remaining vector elements, keeping them together as they proceed through the pipeline. To enable performing operations on multiple keys, the input logic supports reordering fields in a record by de-queueing the input FIFO out of order. Throughout the database logic, provisions are made for partial lists using valid and done signals: it is not possible to determine beforehand the number of elements in a list, so the hardware must dynamically adapt.

3.1.4 Plasticine for Networking

Plasticine has low latency due to its fine-grained pipelined computation; furthermore, Plasticine's latency is deterministic because there are no back-ground tasks to delay processing. In Phase 1, we explored specializing Plasticine's architecture for inference, with minimal hardware overhead, by shrinking the size of on-chip memories, eliminating DRAM controllers, and reduced precision [26]. This reduced architecture could function inside a high-performance network switch and evaluate ML models on each packet with minimal added latency, supporting applications like anomaly detection.

3.2 Phase 2 and Beyond

In Phases 2 of Plasticine II development, we will further specialize Plasticine's compute fabric for more complicated applications, with a focus on sparsity. Section 3.2.1 discusses the hardware sup-port that we use to facilitate sparse accesses, including support for dynamically resolving bank conflicts and supporting applications, like push-based graph processing, that may have unbankable read-modify-write accesses. Section 3.2.2 then discusses how we will use data, such as that from performance counters, to optimize our sparse applications through a continuous re-compilation process, building on our existing work. Section 3.2.4 discusses our proposal for enabling multi-chip support (not evaluated in our performance metrics), and Section 4 discusses the process we use to estimate the performance and energy benefits of our Phase 2 optimizations.

3.2.1 On-Chip Sparsity

Support for on-chip sparsity is a key driver of our Phase 2 evaluation effort. Currently, Plasticine is able to deliver its performance due to the large amount of on-chip memory bandwidth from its user-configured scratchpads. To assist the user, Spatial banks these scratchpads for parallel accesses: up to 16 accesses per PMU, per cycle, for a total of 4 TBps of aggregate memory band-width to 16 MB of memory. This banking resolves both *structural hazards* (two readers cannot access the same bank in the same cycle) and i (parallelization and pipelining leading to incorrect program execution). However, because banking relies on static analysis of accesses to memory (based on loop iterators), it cannot be applied to dynamic applications, which access memory in-directly through random pointers. Instead, new techniques will be required to resolve these hazards and maintain bandwidth for sparse applications.

Structural Hazards To resolve structural hazards, we will add a dynamic dependency resolution unit to assign memory accesses to banks, shown in Figure 11. In the simplest case, only one vector of requests (16 requests) is able to be assigned to memory banks; however, because the banks are random, there is likely to be an approximately 3x slowdown due to bank conflicts. Ideally, accesses would be allowed to bypass each other for bank assignment: if request vector A has several accesses to bank 0, then request vector B should be able to start executing before A completes. How-ever, provisioning a full crossbar limits the number of simultaneous vectorized requests that can be considered, because area and energy increase super-linearly the number of requests. However, if a restricted crossbar is used, this limit is avoided: an allocator selects pending requests to maximize throughput under structural constraints: one re-quest per bank, per cycle, and one or two requests per input lane, per cycle. This will decrease the area and timing

penalty of vectorized sparse memory accesses, with a minimal performance penalty.

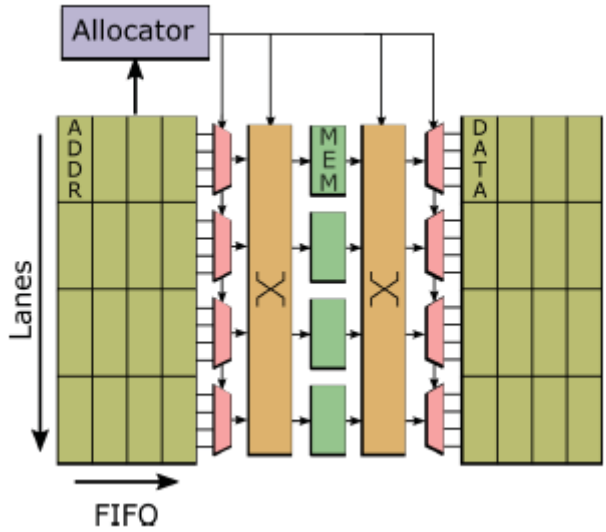

Figure 11: The Allocation Pipeline for Plasticine II Memory Accesses

Data Hazards The other challenge with on-chip sparsity revolves around *memory consistency*: ensuring that, when accesses perform at each memory, they do so in a manner that programmers can understand. Achieving high performance on spatial architectures requires a significant amount of memory access reordering: when multiple iterations of an inner loop are pipelined, the beginning of each loop is executed before the tail of previous loops have completed. Similarly, when loops are executed with fine-grain pipelining, later iterations of the loop begin before previous iterations have completed. For Plasticine I, this is not a concern: the compiler's banking pass allows it to determine that memory accesses will not conflict, and reorder them.

However, for Plasticine II's sparse memory sup-port, banking will not be possible, and new algorithmic, compilation, and hardware techniques are necessary. For some algorithms, like machine learning training, no compiler or hardware modifications are necessary: our prior work has shown that these applications are inherently tolerant of memory ordering violations because they are con-vergent [27]. However, other applications, like inference and packet processing, are not convergent: if an application specification requires that it accurately count packets, then the application cannot drop any updates.

We propose to solve this problem with hardware-assisted memory locking. Memory locking is a well-studied technique for ensuring fast parallel accesses to shared data, but has a few well-known pitfalls [28, 29]. Notably, multiple readers and writers can conflict, especially when a single global lock is used to arbitrate access to memory. Additionally, the need to dedicate CPU cycles to locking adds to program runtimes, and the need to move the locked data between caches before modifying it adds further overhead. Our proposed memory locking solution is *fine-grained* and *hardware-assisted*. It will support the following features:

- Flexible locking granularity
- Fully vectorized: one lock taken and released per vector lane each cycle
- Multiple locks can be taken by each vector lane, with hardware-resolved deadlock
- Lanes that fail to gain memory locks can either be stalled or invalidated and retried (thus avoiding stalls, for applications that can run out of order)
- Reader/writer locks to efficiently support imbalanced read/write access patterns

We will use these primitives to write support for a variety of user-level algorithms, including:

- Dynamic memory allocation (malloc): locking allows operations, like adding blocks to the free list, to be made atomic.
- Atomic variables: locking allows sparse reductions, like incrementing a data-dependent counter, to be made atomic.
- Consistent data structures: as an extension of malloc, we will use atomic memory-memory operations to support a wider variety of data structures.

3.2.2 Increased Dynamism

Plasticine I is reconfigurable at the chip level to support a variety of applications. However, this is insufficient for future, more dynamic applications: these will have more infrequently-active code and more variability in which parts of the chip are active. To support a large amount of inactive code (such as that found in data pre-processing), we will add time-sharing to PCUs. In Phase 2, instead of having a single configuration, each PCU will have a small number (e.g., 4) of configurations, and will dynamically alternate between them. The compiler will use either static analysis or profiling with performance counter to schedule different configurations to share the hardware resource. Furthermore, the current Plasticine I reconfiguration fabric is a single-bit network chain, taking tens of thousands of cycles to fully clock in. To adapt to varying statistics (size, sparsity) of incoming data overtime, Plasticine needs rapid reconfiguration that varies implementation based on data characteristics. Therefore, we will split this chain into several smaller chains and explore hardware techniques for dynamically compressing the bit-stream to decrease the time required to reconfigure Plasticine. We can also overlap reconfiguration with execution to hide reconfiguration latency.

3.2.3 Reconfigurable Cache

Plasticine I's distributed and banked on-chip scratchpads provide a maximum of 4TB/s read and write bandwidth for applications with dense accesses; we capture the benefits of spatial and temporal locality for these applications with tiling. Applications with sparse random accesses cannot be tiled, but they can use a cache that captures both spatial and temporal localities. In Plasticine II, we will add a configurable cache by composing and reconfiguring exists scratchpads in PMUs. Currently, to access DRAM, the user must tile the pro-gram and use an explicit DRAM transfer to load or gather data. We will introduce a new data structure in Spatial, `Cache`, which provides the same interface as `DRAM` but caches DRAM accesses on-chip.

A key challenge is how to use existing distributed on-chip resource to compose a cache that has parameterized capacity, associativity, and block size. Figure 5 shows how we will achieve this goal with a combination of hardware and compiler support. In Figure 5(a), we use the left set of PMUs to stores the tags and the right set to store the data blocks. Upon receiving a DRAM request, the first few bits of the index will be used to locate the PMU that stores the tag.[1] Next, the request will be forwarded to the corresponding PMU over the global network. Tags for different associative ways are stored in different memory banks to allow parallel lookup in a single cycle. If the request misses, the PMU will send the request to DRAM through the global network. Otherwise, the request is sent to the PMU storing the data blocks for the set that hit. The configuration in Figure 5(b) reduces associativity but increases the number of sets stored in one PMU, allowing multiple accesses to search a single PMU in parallel. We can also increase associativity beyond 16 by using banks from multiple PMUs. Figure 5(c) shows mapping both tags and blocks on banks within a single PMU when using small associativity and block size, which provides a single-cycle access latency. When the user declares a `Cache(dram, ways, sets, blksize)`, the compiler allocates required PMUs and sets up their connections. These PMUs will be placed and routed with the other parts of the program graph onto the accelerator array. With this approach, we allow users to trade off the amount of resources spent on caches vs. scratchpads based on application characteristics.

3.2.4 Chip-Crossing Programs

Multi-chip designs will increase bandwidth and allow Plasticine II to solve larger problems. Many of our current applications are memory bandwidth-bound, including database table scans. Although the move to a split HBM/DDR4 memory system that we are exploring with the database work will add memory bandwidth, there is still a limit to the amount of memory and compute band-width that can fit on a single chip. In Phase 2, we will evaluate off-chip network designs, including topologies comprising multiple Plasticine chips: in addition to increasing the total amount of memory bandwidth, this will allow us to break the band-width limitations of planar scaling identified in [9].

3.2.5 Datacenter Operation

In Phase 2, we will extend the current work for Plasticine-based networking, moving from a switch-centric model to include both switch-centric and Plasticine-centric configurations. The switch-centric processing model will continue to provide line-rate ML to accelerate networking applications, with a focus on improving the efficiency, performance, resiliency, security, and scalability of datacenter and wide area networks. The Plasticine-centric configuration will emphasize re-moving CPUs from the data analytics loop: packets will arrive from the network, be parsed by Plasticine, and be replied to. This will require solving a variety of integration challenges, including packet parsing and ensuring security, but will al-low a greater degree of heterogeneous data processing: applications will be able to add additional resources, including frequent access to remote storage, into the critical path.

[1]The DRAM and Cache data structures in Spatial represent logical arrays stored in DRAM with disjoint address spaces. Both are indexed with relative 32-bit offsets within the arrays, which are translated to an absolute 64-bit address be-fore being sent to DRAM.

4. Evaluation

4.1 Architecture

The architecture that we model for our evaluation is Plasticine II as of Phase 1, with additional features to handle sparse applications as described in Section 3.2.1 and Section 3.2.3. The accelerator is running at 1 GHz in 10 nm technology with a projected area footprint of around 100 mm^2 to 150 mm^2. We model HBM and DDR4 off-chip memory with 900 GB/s and 100 GB/s bandwidth, respectively. At the fringe, parallel access streams are supported by 64 off-chip memory interfaces, each of which is coupled with an address generator that can be configured to perform dense or sparse loads and stores. The address generator has a reorder buffer with 1024 entries to reorder off-chip accesses within each stream. For sparse applications, we reconfigure roughly half of the PMUs into a 30 MB, 16-way set-associative cache with a block size of 64B and four-cycle access latency.

4.2 Simulation

We use two different simulation methodologies. For dense deep learning applications, the key kernels (e.g., convolutions) have high compute density; we assume performance is bottlenecked at these compute-heavy kernels. We use a performance model that statically analyzes the mapping of the computation to the Plasticine architecture by considering the number of operations performed and data fetched. The analyzer maps DNN models, expressed as PyTorch compute graphs, to Plasticine and reports hardware utilization, through-put, and power consumption. The accuracy of the analyzer has been verified against DNN models that have been executed on Plasticine's RTL. Next, we use these numbers to compute performance for various data sizes.

For the other applications (e.g., sparse linear algebra and graph applications), we assume that the key kernels do not have the compute density to saturate Plasticine II. Therefore, performance is bottlenecked by data accesses from off-chip memory. To model these applications, we ex-tend Python-based DSLs' native data containers and operators with features to generate memory access traces. A trace records the size and type of data access, as well as its dependency on previous traces. Using this information, Tungsten, our cycle-accurate Plasticine simulator, simulates off-chip accesses and the cache to provide performance and power estimates.

We only simulate the data processing part of the applications. We assume that the data are preprocessed and loaded into memory before the application starts. The reference implementations make the same assumption to measure the applications' runtime performance.

For each application, we simulate on a subset of the complete dataset and extrapolate the performance on the whole dataset. For deep learning applications, we run inference or training for one batch of data, calculate the throughput, and extrapolate the performance on the complete dataset. For the LGC implementations, the program execution time roughly scales linearly with the number of seeds. We simulate the LGC implementations with one seed and scale the performance by the total number of seeds. We use the same method-ology to extrapolate performance for Sinkhorn WMD and IPNSW.

4.3 Discussion

Plasticine II provides up to 400–3200x speedups compared to the CPU baseline for representative training and inference applications, respectively. The convolutional neural network has a relatively small speedup (2.4x for training and 22.6x for inference). The small speedup mostly attributes to the compute FLOPS underutilization due to misalignment between the kernel size and CIFAR-10 image sizes dataset with the SIMD lane width. For most of the compute-intensive applications, we achieve roughly 100-170GOPS/Watt. We expect the GOPS/Watt to be doubled for training with `fp16` and quadrupled for inference with `fp8` after we introduce subword SIMD in Phase 2. Percentage to theoretical peak assumes present of this optimization.

For most graph and sparse applications, we achieve a 30-300x speedup compared to the CPU baseline, with the only exception in LGC/ISTA. The key kernel in LGC/ISTA is an iterative, sparse gradient descent algorithm with a loop-carried dependency, which prevents us from exploiting coarse-grained parallelism. How-ever, LGC/pr nibble, a parallel LGC algorithm, achieves a 30x speedup compared to the CPU baseline.

We do not provide the projected GOPS/W for the graph applications. It is hard to estimate the projected GOPS/W without knowing the input data layout, and the provided datasets are too small to saturate the memory bandwidth of HBM. We expect additional speedup compared to CPU when using larger graph datasets.

References

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

[2] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 389–402, June 2017.

[3] S. Hadjis and K. Olukotun, "Tensorflow to cloud FPGAs: Tradeoffs for accelerating deep neural networks," in *Proceedings of the 29th International Conference on Field Programmable Logic and Applications, FPL`19*, IEEE, 2019.

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

[5] A. Cilardo and L. Gallo, "Interplay of loop unrolling and multidimensional memory partitioning in hls," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, (San Jose, CA, USA), pp. 163–168, EDA Consortium, 2015.

[6] D. Koeplinger, R. Prabhakar, Y. Zhang, C. De-limitrou, C. Kozyrakis, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 115–127, IEEE Press, 2016.
[7] B. Bodin, L. Nardi, H. Wagstaff, P. H. J. Kelly, and M. O'Boyle, "Algorithmic performance-accuracy trade-off in 3d vision applications," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (IS-PASS)*, pp. 123–124, April 2018.
[8] L. Nardi, D. Koeplinger, and K. Olukotun, "Practical design space exploration," *CoRR*, vol. abs/1810.05236, 2018.
[9] Y. Zhang, A. Rucker, M. Villim, R. Prabhakar, and K. Olukotun, "Scalable interconnects for reconfigurable spatial architectures," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2019.
[10] T. Zhao, Y. Zhang, and K. Olukotun, "Serving recurrent neural networks efficiently with a spatial accelerator," in *Proceedings of the 2nd SysML Conference*, 2019.
[11] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium* on, pp. 1–12, IEEE, 2017.
[12] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," *arXiv preprint arXiv:1803.04014*, 2018.
[13] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, *et al.*, "Ese: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 75–84, ACM, 2017.
[14] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, et al., "A configurable cloud-scale DNN processor for real-time AI," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pp. 1–14, IEEE Press, 2018.
[15] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural net-works," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.
[16] S. Narang and G. Diamos, "Baidu deep-bench," *GitHub Repository*, 2017.
[17] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun*. ACM, vol. 59, pp. 56–65, Oct. 2016.
[18] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky, "Optiml: An implicitly parallel domain-specific language for machine learning," in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, (USA), pp. 609–616, Omnipress, 2011.
[19] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001–.
[20] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Good-fellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Man´e, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V.

Vasudevan, F. Vi´egas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.
[21] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in Pytorch," 2017.
[22] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.),(Pasadena, CA USA), pp. 11 – 15, 2008.
[23] J. Leskovec and R. Sosic, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.
[24] A. Ratner, S. H. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Re, "Snorkel: rapid training data creation with weak supervision," *The VLDB Journal*, Jul 2019.
[25] M. Villim, A. Rucker, Y. Zhang, S. Liu, and K. Olukotun, "Gorgon: Accelerating machine learning from relational data (in sub-mission)," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, IEEE, 2020.
[26] T. Swamy, A. Rucker, M. Shahbaz, N. Yad-wadkar, Y. Zhang, and K. Olukotun., "Taurus: An intelligent data plane," in *P4 Workshop*, (Stanford, CA USA), 2019.
[27] C. De Sa, K. Olukotun, and C. Re, "Ensuring rapid mixing and low bias for asynchronous gibbs sampling," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pp. 1567–1576, JMLR.org, 2016.
[28] M. W. Blasgen, J. Gray, M. F. Mitoma, and T. G. Price, "The convoy phenomenon," *Operating Systems Review*, vol. 13, no. 2, pp. 20–25, 1979.
[29] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, (New York, NY, USA), pp. 289–300, ACM, 1993.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACRONYM	DESCRIPTION
AG	address generator
AI	artificial intelligence
ASIC	application-specific integrated circuit
CGRA	coarse grain reconfigurable array
CPU	central processing unit
DB	database
DDR3	double data rate type 3
DNN	deep neural network
DRAM	dynamic random-access memory
DSE	design space exploration
DSL	domain-specific language
FIFO	first-in, first-out
FLOPS	floating-point operations per second
FPGA	field-programmable gate array
FSM	Finite State Machine
FU	functional unit
GPU	graphics processing unit
HBM	high-bandwidth memory
IO	input/output
IR	infrared
ML	machine learning
NoC	network on chip
PaR	place and route
PCU	pattern compute unit
PMU	pattern memory unit
RDU	reconfigurable dataflow unit
RNN	recurrent neural network
SDH	Software Defined Hardware
SIMD	single instruction multiple data
SRAM	static random-access memory
TFLOPS	tera floating-point operations per second
WAR	write after read
WAW	write after write