2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00085

SARA: Scaling a Reconfigurable Dataflow Accelerator

Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, Kunle Olukotun
Stanford University

***Abstract*—The need for speed in modern data-intensive workloads and the rise of "dark silicon" in the semiconductor industry are pushing for larger, faster, and more energy and area-efficient architectures, such as Reconfigurable Dataflow Accelerators (RDAs). Nevertheless, challenges remain in developing mechanisms to effectively utilize the compute power of these large-scale RDAs.**

To address these challenges, we present SARA, a compiler that employs a novel mapping strategy to efficiently utilize large-scale RDAs. Starting from a single-threaded imperative abstraction, SARA spatially maps a program onto RDA's distributed resources, exploiting dataflow parallelism within and *across* hyperblocks to saturate the compute throughput of an RDA. SARA introduces (a) compiler-managed memory consistency (CMMC), a control paradigm that hierarchically pipelines a nested and data-dependent control-flow graph onto a dataflow architecture, and (b) a compilation flow that decomposes the program graph across distributed heterogeneous resources to hide low-level RDA constraints from programmers. Our evaluation shows that SARA achieves close to perfect performance scaling on a recently proposed RDA—Plasticine. Over a mix of deep-learning, graph-processing, and streaming applications, SARA achieves a 1.9× geo-mean speedup over a Tesla V100 GPU using only 12% of the silicon area.

***Index Terms*—RDA, CGRA, Plasticine, Scalability, Domain-Specific Compiler**

I. Introduction

With the end of Dennard Scaling and the decline of Moore's Law, single-core CPU performance has plateaued [12]. Multicore scaling is also approaching its limit due to the power wall [14], [21]. Thus, to sustain the economics of process scaling, recent efforts leverage applications' domain knowledge to enable performant and power-efficient hardware accelerators [9], [19], [21], [24].

Among these, Reconfigurable Dataflow Accelerators (RDAs) [41] are an emerging class of large-scale Coarse-Grained Reconfigurable Arrays (CGRAs) designed for the compute intensity of modern analytical workloads. With reconfigurable datapath, RDAs can exploit multiple-level of data and pipeline parallelism, significantly improving their compute throughput [7], [30]. Moreover, with coarse-grained reconfigurable blocks, RDAs are more energy-efficient and resource-dense than Field Programmable Gate Arrays (FPGAs) [30]. Unlike classic CGRAs [16], [33], [45] that often tightly integrate with a general-purpose processor, RDAs are designed as stand-alone and server-scale accelerators for data-intensive analytical workloads. With efficiency comparable to domain-specific accelerators [9], [19], RDAs have more flexible datapaths that support a wide range of data-parallel applications across various domains.

At a high level, RDAs are highly distributed on-chip systems with reconfigurable units. Unlike a multiprocessor system, RDA's compute units are statically configured. This eliminates the dynamic execution overhead in a processor core and improves compute density. Furthermore, RDAs use distributed, software-managed scratchpads rather than a hardware-managed shared memory cache, significantly reducing hardware complexity with better and predictable memory performance.

While highly efficient, the size of RDAs demands a new mapping strategy to fully utilize their potential. Since classic CGRAs are much smaller scale, conventional CGRA mapping schemes [16], [33], [44] only exploit one-level of dataflow execution within a hyperblock—a basic block with internally convergent non-looping control flow [31]. The accelerator executes one hyperblock at a time and reconfigures across hyperblock. However, this strategy no longer suffices for large-scale RDAs, as a single hyperblock provides insufficient instruction-level parallelism (ILP) to saturate RDAs' compute power. Furthermore, while hierarchical datapaths and explicitly managed scratchpads make RDAs scalable, they also incur a number of programmability challenges. Programmers must account for low-level resource constraints and manually orchestrate data movement across distributed memory.

To efficiently utilize RDAs while improving their programmability, we introduce SARA, a compiler that generates highly scalable RDA programs from a single-threaded imperative abstraction. We introduce Compiler-Managed Memory Consistency (CMMC), a control paradigm that constructs multiple-level of dataflow execution within and *across* hyperblocks. This mapping strategy enables SARA to exploit ILP over a large window of a control-flow graph (CFG) to saturate RDA's compute throughput. Furthermore, SARA systematically handles hardware constraints with resource virtualization. By automatically partitioning a program to

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7865, and National Science Foundation (NSF) under the award CCF-1563078, 1937301, 1563113, and 2028602. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA), National Science Foundation, or the U.S. Government. This research is also supported in part by members and other supporters of the Stanford DAWN project.

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00085

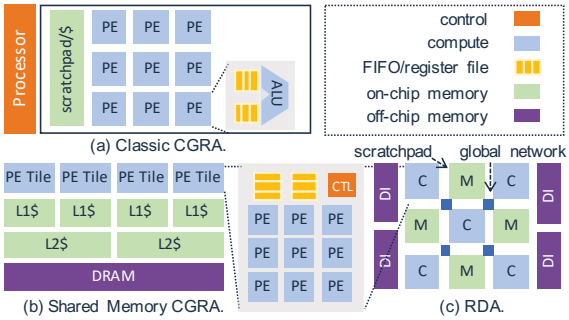


Fig. 1: Compute and memory organization of CGRAs.

satisfy the complex constraints of an RDA with heterogeneous resources, SARA substantially reduces the programmer burden to manage RDAs.

We implement the proposed techniques in SARA and target a recently published RDA—Plasticine [41]. Our evaluation shows that SARA achieves a perfect performance scaling until exhausting the on-chip resources for compute-bound applications, or HBM bandwidth for memory-bound applications. We demonstrate a $4.9\times$ geo-mean speedup over a vanilla Plasticine compiler [41], and a $1.9\times$ geo-mean speedup against a Tesla V100 GPU, which is $8\times$ larger than our target RDA.

In summary, we make the following contributions:

- A novel mapping strategy that exploits multiple levels of pipeline and loop-level parallelism, within and *across* hyperblocks, in a streaming dataflow fashion, using CMMC.
- A compilation workflow to systematically handle RDA constraints and algorithms for program partitioning.
- A quantitative evaluation on the scalability of SARA (§IV-A), the effectiveness of our compiler optimizations (§IV-B), and a comparison to an existing compiler (§IV-C) and an accelerator (§IV-D).

Next, we begin with a detailed discussion on the architectural features of RDAs that improve their scalability, and how these features impact the compilation strategies required to exploit this scalability.

II. RDA Overview

A. Types of RDAs

The primary distinction between classic CGRAs and an RDA is the scale of the architecture. As shown in Fig. 1a, a classic CGRA [15], [16], [20], [33], [38], [45] comprises a grid of processing elements (PEs), each containing an ALU and some local buffers. The PE array can be configured to execute a block of *control-free* instructions in a dataflow fashion. When compiling to a classic CGRA, a compiler inspects the dataflow graph (DFG) within a hyperblock, and generates a schedule and a placement to map the DFG onto the PE array. Most CGRAs are tightly integrated with a co-processor or a centralized scheduler that manages and reconfigures the CGRA across hyperblocks at runtime.

An RDA, at a high level, is a hierarchical and distributed reconfigurable architecture (Fig. 1c). An RDA comprises hundreds of compute and memory units, each containing a set of processing units and/or scratchpads. Modern RDAs are built for compute- and data-intensive analytical workloads, where each unit has compute density comparable to a traditional CGRA. A modern RDA [36], [41], [50] may contain hundreds of such units and is 10x–300x more OPS intensive than a classic CGRA [16], [33], [38]. The scale of RDA thus motivates a number of innovations in compilation strategies.

a) ILP beyond a hyperblock

First and foremost, RDA demands an effective strategy to scale performance across hundreds of distributed reconfigurable units. Traditional mapping strategy adopted by most HLS compilers only exploit SIMD parallelism and one-level of pipeline parallelism within a hyperblock [34], [56]. Some techniques statically unroll loop bodies to expand the scope of the dataflow graph [16], [44]. However, a modern RDA contains over 10k function units. ILP within a hyperblock is largely insufficient to saturate the compute capability of a large-scale RDA.

To effectively utilize an RDA, SARA adopts a novel mapping strategy that spatially pipelines a control-flow graph (CFG) over RDA's distributed resources. This is in stark contrast to Von Neumann-based architectures and classic CGRAs, which execute hyperblocks of a CFG in time rather than in space. More concretely, SARA allocates *dedicated* compute units for each hyperblock and executes them in a dataflow fashion. SARA then wires up synchronization across hyperblocks to ensure the memory access order across hyperblocks matches the program order expected by the CFG. By enforcing that the memory order is consistent with a sequentially executed program, SARA can maintain correctness while exploiting ILP over a large window of the CFG.

b) Nested loop-level parallelism

To maximize the throughput of the overall pipeline, it is crucial to balance the pipeline stage delays, for each level of the coarse-grained pipeline. Our front-end language, Spatial [26], captures an independent parallelization factor for each loop nest in the CFG. When a user parallelizes the innermost loop, SARA vectorizes the computation along the SIMD datapath of Plasticine. When parallelizing the outer loops, SARA spatially unrolls the loop body across distributed compute units, forming a larger compute graph. By combining multiple levels of pipeline parallelism and loop-level parallelism in an arbitrarily nested CFG, SARA can effectively scale the performance of a small compute kernel over a large number of resources. In contrast to a multiprocessor that exploits thread-level parallelism across distributed units, SARA exploits multiple levels of pipeline parallelism across compute and memory units, which significantly improves the overall compute throughput.

c) Flexible control constructs

The scale of RDAs comes with the cost of a long reconfiguration overhead. In contrast to 10s of cycles in a classic CGRA, our target RDA (Plasticine) takes 10s of μs for each reconfiguration. In classic CGRAs, the accelerator often relies on the co-processor or the scheduler to handle complex control flow. In RDAs, the long reconfiguration time

requires the accelerator to operate independently to minimize host intervention overhead. Hence, our spatially pipelined CFG must support dynamic control flow, such as loops with dynamic iterations and branch statements, to ensure the size of a CFG that can be spatially mapped on an RDA is only limited by the amount of on-chip resources, rather than by unsupported control flow constructs.

B. Hierarchical Reconfigurable Fabric and Network On-Chip

a) Distributed control flow

To scale to a large number of units while maintaining a fast clock frequency, RDAs have an interconnection network similar to a traditional Network-on-Chip that introduces dynamic network delays on both data and control paths. However, spatially pipelined execution is particularly sensitive to synchronization overhead, as pipeline bubbles can significantly impact the overall throughput [55]. Hence, orchestration of the hierarchical pipeline must adopt a distributed control flow scheme to minimize control overhead.

b) Composability

Unlike FPGAs with flat reconfigurable fabric, RDAs have a hierarchical datapath that significantly improves their clock frequency and scalability. Nevertheless, hierarchy introduces additional compilation restrictions. Specifically, a hyperblock may contain more operations that do not fit in a compute unit, and a data structure may exceed the capacity of individual MUs. Unlike a hardware synthesis compiler that can freely size a processing core and allocate block RAMs as needed when targeting an FPGA, an RDA compiler must address RDA's resource constraints, composing and virtualizing these distributed units to form larger logical resources.

C. Distributed and Streaming Memory Interfaces

RDAs have distributed and explicitly managed scratchpads to enhance the capacity and bandwidth of on-chip memories. Most classic CGRAs use a centralized memory interface that is sometimes shared with a host processor [15], [16], [20], [33], [38], [45]. Other larger-scale CGRAs attach a shared memory cache to distributed compute units [5], [15], [20], shown in Fig. 1b. Both memory organizations do not scale in performance for a large number of units. RDAs have distributed scratchpads and DRAM interfaces (DIs), shown in Fig. 1c. The scratchpads have disjoint address spaces. While all DIs have access to a global off-chip address space, the hardware provides no ordering across DIs. All memory interfaces serve requests in a streaming in-order fashion and provide an acknowledgment for each request for the software to handle memory order across request streams. While this memory abstraction substantially improves capacity and bandwidth, it leaves the burden of memory management to the software; a user must explicitly manage off/on-chip data transfer and manually partition data across distributed scratchpads.

a) Consistency

A major difference in RDAs memory abstraction is the lack of one-to-one correspondence between a compute core and a memory interface, like in a cache system. RDA compute unit can directly issue requests to *any* on-chip memory unit and DRAM interface. Consequently, a memory interface can no longer observe the program order across concurrent request streams from a spatially pipelined CFG. Hence, the software (in our case, the compiler) must explicitly enforce consistency across multiple request streams. This is managed by the hardware in a traditional memory system.

b) Scaling memory bandwidth with parallel computing

When parallelizing the compute of a spatially pipelined CFG, the memory bandwidth of intermediate data must scale accordingly to avoid stalling the pipeline. SARA achieves this by sharding a logical tensor across distributed memory units, effectively scaling memory bandwidth with parallelization. This is very different from the conventional shared memory hierarchy that relies on hardware to manage coherence when compute is parallelized, which is hard to scale in performance.

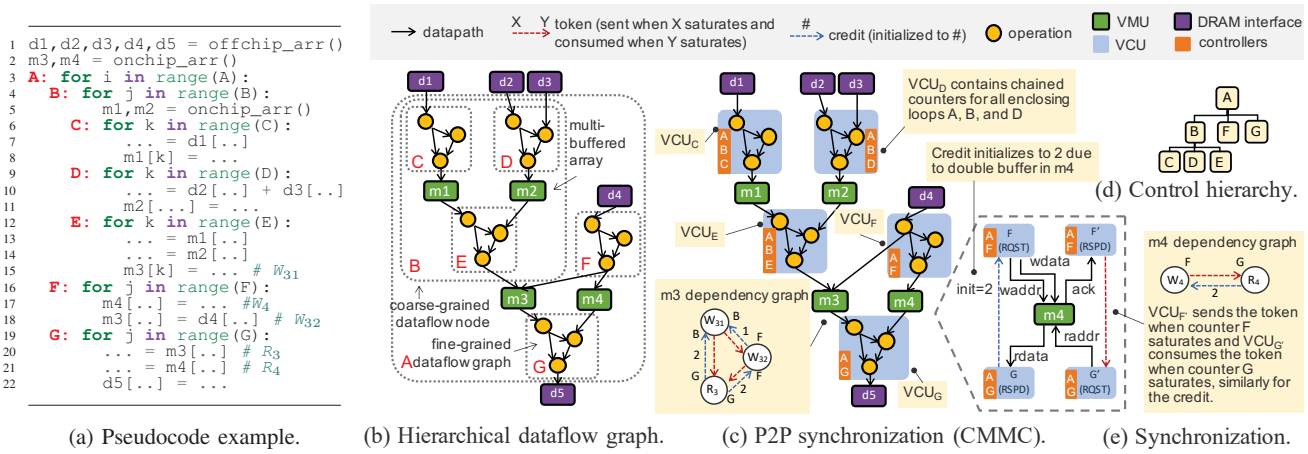
III. Design of SARA

This section details how SARA converts a nested CFG from a single-threaded imperative abstraction into a hierarchical DFG that spatially pipelines the CFG across distributed RDA units. Our front-end language is Spatial [26], a domain-specific language with a nested loop abstraction primarily targeting FPGAs. We extend Spatial with our back-end compiler, SARA, to target the Plasticine RDA. Fig. 3 shows SARA's two-phase process to target Plasticine: a **synthesis** phase that generates a hierarchical dataflow graph (DFG), and a **placement and routing (PnR)** phase that maps the DFG onto the physical network. We omit the discussion on PnR as it is well studied in previous CGRA mapping work [34], [39], [56].

The synthesis phases generate a two-level hierarchical DFG, which we refer to as a Virtual Unit Dataflow Graph (VUDFG). The top-level graph maps across RDA's distributed physical units (PUs), and the inner DFG maps to the reconfigurable fabric within each PU.

SARA synthesizes the dataflow graph in two steps. First, SARA performs **imperative to dataflow lowering** (§III-A). This step allocates a virtual compute unit (VCU) for each hyperblock and a virtual memory unit (VMU) for each on-chip data structure that holds the intermediate results across hyperblocks (Fig. 2c). Next, SARA allocates control handshakes across virtual units (VUs), such that their execution order conforms to the schedule of the original CFG. At this point, the virtual units (VUs) have no notion of the resource constraints of PUs. Next, SARA takes a specification of PU constraints and performs **abstract resource mapping** (§III-B), subdividing oversized VUs into smaller VUs and merging small VUs into larger VUs to reduce resource fragmentation. SARA first assigns each VU to one particular PU type, and then places and routes them onto the physical network.

Lastly, §III-C enumerates **optimizations** applied throughout these steps. These optimizations either reduce the resource cost of the VUDFG, or alleviate potential performance bottlenecks in a streaming pipeline.



(a) Pseudocode example. (b) Hierarchical dataflow graph. (c) P2P synchronization (CMMC). (d) Control hierarchy. (e) Synchronization.

Fig. 2: (b) a 3-level hierarchical DFG for (a): 1) pipelined DFGs for the innermost loops 2) a second level DFG {C, D, E} pipelined across iterations of loop B, and 3) the top-level DFG {B, F, G} pipelined across iterations of loop A. (c) VUDFG with a P2P control scheme. (f) Control hierarchy for (a). (e) Detailed control setup for $\texttt{VMU}_{m3}$ between $\texttt{VCU}_F$ and $\texttt{VCU}_G$.

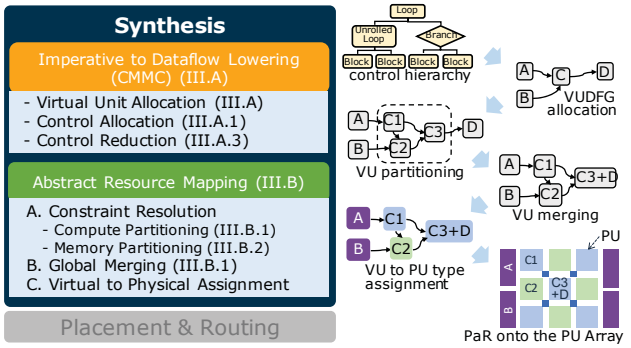


Fig. 3: SARA's compilation flow.

A. Imperative to Dataflow Lowering

Starting from a nested imperative abstraction, SARA generates a hierarchical dataflow graph that executes the CFG in a coarse-grained pipelined fashion. This mapping strategy is inspired by hierarchical pipelining, a concept first introduced for FPGAs in DHDL [40]. Concretely, Fig. 2a shows a 3-level nested loop with five basic blocks. At the innermost level, operations of each loop can execute in a fine-grained pipelined fashion. Furthermore, with dedicated resources allocated for each basic block, we can coarse-grained pipeline loop C, D, and E across iterations of loop B, and pipeline loop B, F, G across iterations of the outermost loop A. Each loop can be independently parallelized to balance the stage latency for each level of the coarse-grained pipeline. A perfectly balanced pipeline (not always feasible) can execute the entire CFG concurrently, maximizing the compute utilization. This execution model effectively scales the performance of a small kernel over on-chip resources of a reconfigurable accelerator, and is well suitable for data-analytical workloads that often contain deeply nested control flow.

The FPGA implementation of hierarchical pipelining, however, can be inefficient on RDAs. The DHDL compiler adopts a hierarchical finite-state machine (FSM) represented by the control hierarchy shown in Fig. 2d; each FSM corresponds to a loop that schedules the inner-loop FSMs using *enable* and *done* handshake signals. While these signals travel in a single cycle on FPGAs, they take 10s of cycles on an RDA network due to RDA's higher clock frequency, incurring significant pipeline bubbles. Furthermore, unrolling loop B with three stages by 10x, for example, creates a 30-to-1 synchronization hotspot, potentially failing PnR due to RDA's limited routing resources.

To address these problems, we introduce *Compiler-Managed Memory Consistency (CMMC)*, a control paradigm for hierarchical pipelining in a distributed setting. Instead of hierarchical handshakes, CMMC adopts a peer-to-peer control scheme with minimum overhead and thus ensures scalability.

In the rest of this section, we first describe the mechanism of CMMC in §III-A1. §III-A2 discusses the handling of data-dependent control flows, such as branches and do-while loops. Lastly, §III-A3 presents an analysis that reduces synchronization while preserving correctness.

1) Compiler-managed memory consistency (CMMC)

As a start, SARA maps each hyperblock (C, D, E, F, and G) to a virtual compute unit (VCU) and maps each on-chip data structure to a virtual memory unit (VMU) (Fig. 2c). Each VMU only connects to a set of VCUs that produces and consumes the data structure stored in the VMU. Next, SARA allocates control handshakes across VUs to enforce the schedule of a hierarchical pipeline.

Instead of orchestrating the execution order of VCUs, the idea of CMMC is to enforce the memory access order for *individual* data structures. More concretely, the CFG expects `m1` to be written by $\texttt{VCU}_C$ for C times, before being read by $\texttt{VCU}_D$ for D times, repeating A×B times. Hence, SARA enforces this order between the two request streams from $\texttt{VCU}_C$ and $\texttt{VCU}_E$ to $\texttt{VMU}_{m1}$. This synchronization is localized

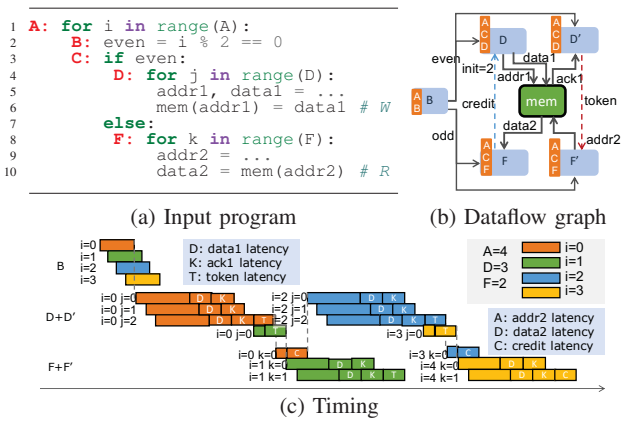
among VCUs that access the same data structure. As long as we guarantee all *individual* data structures are updated in a program-consistent order, the final result will be identical to a sequentially executed program.

The result of CMMC is that *only hyperblocks with data dependencies are ordered*. This is a key improvement over both von Neumann architectures and other CGRA mapping techniques, where hyperblocks are executed in time. Prior work [48] unnecessarily orders independent hyperblocks, and even techniques like speculative execution only mildly alleviate the ordering between adjacent hyperblocks. As long as resources permit, CMMC can exploit ILP across the *entire* CFG, which is essential to fully utilize an RDA's compute capability. Currently, we rely on the user or a higher level compiler to segment a CFG that is too big to fit on an RDA.

We use $\texttt{VMU}_{m4}$ to demonstrate how SARA enforces the memory order between $\texttt{W}_4$ and $\texttt{R}_4$. As shown in Fig. 2c, SARA maps all enclosing loop counters into each VCU. $\texttt{VCU}_F$, for example, contains chained counters corresponding to loops F and A. The innermost counter increments every cycle when the VCU is enabled, when all of its input streams and tokens are valid and output streams are ready. To avoid round-trip latency between the VCU and the VMU, SARA partitions each memory access into a *request* and a *response* VCU, as shown in Fig. 2e. For read access $\texttt{R}_4$, SARA maps the address generation in a request $\texttt{VCU}_{G'}$, streaming addresses to $\texttt{VMU}_{m4}$ and streaming data back to the original $\texttt{VCU}_G$. For write access $\texttt{W}_4$, SARA allocates a response $\texttt{VCU}_{F'}$ to accumulate the write acknowledgments. $\texttt{VCU}_{F'}$ only contains the same set of counters as $\texttt{VCU}_F$ with no datapath. In common cases, SARA maps $\texttt{VCU}_{F'}$ and $\texttt{VCU}_{G'}$ to the same Plasticine memory unit that maps $\texttt{VMU}_{m4}$.

A token is just a single-bit pulse treated as a data dependency of the receiver VCU. It can be viewed as a grant passed among accessors to ensure they are executed in the expected program order. Concretely, to enable $\texttt{W}_4$ F times before $\texttt{R}_4$, SARA configures $\texttt{VCU}_{F'}$ to send a token to $\texttt{VCU}_{G'}$ after its local counter F saturates. On the reader side, $\texttt{VCU}_{G'}$ consumes one token when its counter G saturates. When enforcing access A to happen before B, we always send the token from the *respond* VCU of A, to *request* VCU of B, to ensure that B observes the memory effect of A in a remote VMU.

To prevent $\texttt{W}_4$ from overriding $\texttt{R}_4$'s data before consumed, SARA configures a backward token from $\texttt{VCU}_G$ to $\texttt{VCU}_F$, sent when counter G saturates and consumed when counter F saturates. The backward token, also known as the credit [11], is initialized to at least one token at the destination, to break the cyclic dependency. When the initial credit is set to one, the memory order would match to a sequentially executed CFG, and hence guarantees correctness. For certain access patterns, we can relax the credit to be larger than one (1+) to enable the first access multiple times before getting back-pressured, which enables the pipeline execution across accessors. A 1+ initial credit often matches with the VMU's multibuffer depth to track the downstream buffer size. For example, we can double-buffer $\texttt{VMU}_{m3}$ and set the initial credit to 2, if



(a) Input program (b) Dataflow graph

(c) Timing

Fig. 4: An example program with an outer branch statement.

$A(R_4) \subset A(W_4)$, where A is the address span of the accessors for every iteration of their enclosing loop A, The initial credits of m3 in Fig. 2c assumes $A(R_3) \subset \{A(W_{31}) \cup A(W_{32})\}$. We rely on the high-level compiler, Spatial [26], to perform the address analysis to relax the back-pressure strength.

To determine when the token is released from access $\texttt{W}_{31}$ and consumed by access $\texttt{R}_3$, for example, SARA first finds their least common ancestor in the control hierarchy (Fig. 2d), i.e. loop A, and then uses the done of the immediate child ancestors, i.e. loop B for $\texttt{W}_{31}$ and loop G for $\texttt{R}_3$, to drive the push and pop signals at the source and the destination.

To break the cyclic dependency between a VCU and a VMU, each hyperblock needs to be partitioned into two VCUs if the block contains a write access followed by a read access to the same VMU. In such a scenario, the token is sent and consumed for every cycle the VCUs are enabled.

2) Data-dependent control flow

To minimize host intervention overhead, SARA supports flexible data-dependent control constructs on RDAs. It is worth noting that our goal is not to accelerate general-purpose applications. Control-heavy workloads with extensive use of global data structures would perform horribly with CMMC due to excessive synchronization. RDA's primary applications are still data-analytical workloads with relatively static control flow and abundant nested loop-level parallelism.

a) Dynamic loop bounds

The min, stride, and max of a for loop can be data-dependent. SARA maps the computation that generates the bounds into a separate VCU and streams the bound values as data dependencies to VCUs enclosed by the loop—a bound value is consumed every time the loop is completed.

b) Branch

SARA enables spatial pipelining of a control flow graph with outer branches that enclose loops and hyperblocks, which none of the prior work support [44], [48]. SARA SARA also supports branches within a hyperblock using predication [4].

For an outer branch shown in Fig. 4, SARA maps computation under different clauses to different VCUs, which only execute when enabled by the branch conditions. A separate VCU evaluates the conditions and passes them as data

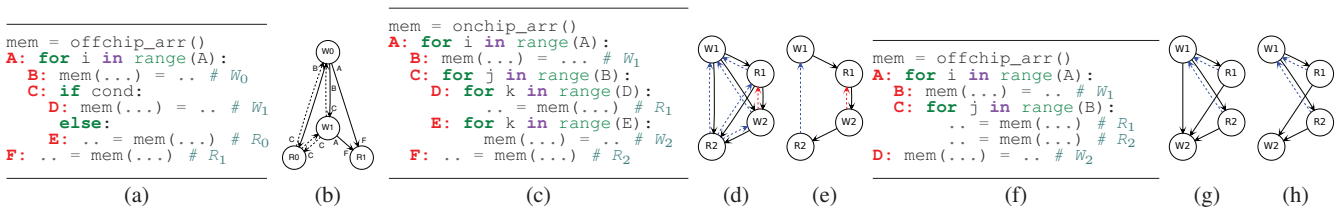


Fig. 5: (b,d,g) dependency graphs for (a,c,f), and (e,h) reduced dependency graphs for (d,g), respectively. Solid edges indicate forward dependencies and dashed edges indicate backward loop-carried dependencies (LCDs). Colors of LCD edges in (d,e,g,h) indicate the associated loops, blue for loop A and red for loop B.

dependencies to VCUs mapping the branch clauses. The input program (Fig. 4a) writes and reads the memory mem on even and odd cycles of the outer loop A, respectively. SARA maps the three hyperblocks in the program to three VCUs, $\texttt{VCU}_B$, $\texttt{VCU}_D$, and $\texttt{VCU}_F$ (Fig. 4b), and splits request and response VCUs for W and R. B generates the branch conditions and sends them to VCUs under the *if* and the *else* clauses.

Fig. 4c shows the timing of the execution. As $\texttt{VCU}_B$ has no dependencies, $\texttt{VCU}_B$ executes the conditions for all iterations of loop A and broadcasts the conditions to other VCUs. SARA sets up tokens to order W and R. If branch C is enabled for the respective clause, $\texttt{VCU}_D$ and $\texttt{VCU}_F$ send and consume the tokens when loop D and F are *done*, respectively. Otherwise, tokens are immediately released to the downstream consumer. In this example, $\texttt{VCU}_D$ and $\texttt{VCU}_{D'}$ skip loop D for i=1,3, and $\texttt{VCU}_F$ and $\texttt{VCU}_{F'}$ skip loop F for i=0,2. By skipping the disabled branches, both *if* and *else* VCUs only execute half of A's iterations. Fig. 4c shows an overlapping execution between the *if* and the *else* clauses. If loop A runs for N iterations and the latencies for loop D and F are L, the total runtime is approximately $\frac{N}{2}L$, which is half of NL—the latency of traditional hierarchical pipelining.

c) Do-while loops

The *do-while* construct is useful to express iterative convergence algorithms or handle an external data stream with a last-bit signal that terminates the execution of the accelerator. A do-while loop works similarly to a for loop with dynamic bounds, except the do-while loop has a much longer initiation interval (II) [42]. The while condition is a data dependency to all VCUs enclosed by the do-while loop, and the condition may also be a function of the VCUs' output. The earliest starting time for the second iteration of the loop is when the while condition is resolved.

3) Control-reduction analysis

Enforcing the order between every pair of accessors is sometimes unnecessary, as synchronizing a subset of the dependencies can preserve the same total ordering. To minimize the allocated tokens, SARA builds a dependency graph between accessors *per data structure*; nodes represent accessors and edges represent dependencies. Next, SARA removes edges that do not relax the ordering across accesses. Lastly, SARA inserts tokens to enforce dependency for every edge in the reduced dependency graph, using the mechanism explained in §III-A1.

a) Dependency graph construction

For each accessor, SARA checks preceding accesses in the program order for forward dependencies, and backward loop-carried dependencies (LCDs) if the two accesses are enclosed by a loop. Because the producer and consumer accessors are pipelined, we need to enforce write-after-read and write-after-write dependency to protect the consumer's data from being overridden by the producer. In Fig. 2a, for example, $\texttt{W}_{4,i+1}$ depends on $\texttt{R}_{4,i}$ because $\texttt{W}_{4,i+1}$ cannot override $\texttt{R}_{4,i}$'s data before it is consumed. For VMU, SARA also enforces read-after-read since Plasticine's PMU can only serve one read request stream at a time. The backward edges in the dependency graph get converted into credits.

Fig. 5b shows an example dependency graph. $\texttt{W}_1$ and $\texttt{R}_0$ have no forward dependency because their LCA controller is branch C, i.e. the two accesses cannot execute simultaneously for the same iteration of loop A. There are, however, LCDs between $\texttt{R}_0$ and $\texttt{W}_1$, preventing reordering of two streams across iterations of A. $\texttt{R}_0$ and $\texttt{R}_1$ are independent because the DRAM interface permits concurrent read streams. $\texttt{R}_1$ and $\texttt{W}_0$ have no LCD as they are not enclosed by a loop.

b) Dependency graph reduction

After graph construction, SARA reduces the dependency edges in two passes, processing forward and backward dependencies separately.

For forward dependencies, SARA performs a transitive reduction (TR) [3] on the forward-dependency graph. TR retains the minimum number of edges in a graph that preserves the connectivity of the original graph, and thus enforces the same memory ordering. For example, edge $\texttt{W}_1$-$\texttt{W}_2$ and $\texttt{R}_1$-$\texttt{R}_2$ in Fig. 5d, and $\texttt{W}_1$-$\texttt{W}_2$ in Fig. 5g are removed during this pass. The forward edges can be reduced with TR because forward dependencies are transitive, i.e. *A* depending on *B* and *B* depending on *C* enforces *A* depending on *C*.

Next, SARA processes the backward dependency graph. A backward edge A-B with X initial tokens can be removed, if there exists an alternative path from A to B, where the new path contains a *single* backward edge representing an LCD of the same loop with the same amount of initial tokens. This is because the initial token for A-B enables B X times before waiting for A, and the alternative path would enforce the same behavior. For example, edge $\texttt{R}_2$-$\texttt{R}_1$ in Fig. 5d is pruned because of path $\texttt{R}_2$-$\texttt{W}_1$-$\texttt{R}_1$, and $\texttt{W}_2$-$\texttt{W}_1$ is subsumed by $\texttt{W}_2$-$\texttt{R}_2$-$\texttt{W}_1$. $\texttt{R}_2$-$\texttt{W}_1$ in Fig. 5d and Fig. 5g cannot be removed

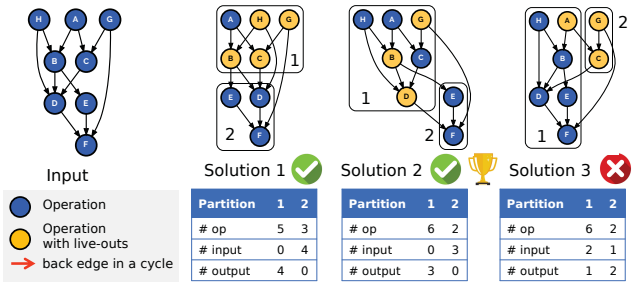


Fig. 6: Compute partitioning example.

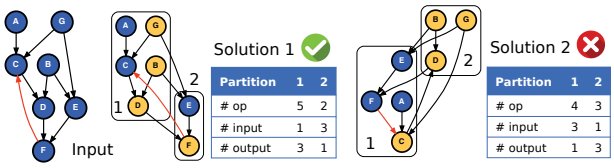


Fig. 7: Compute partitioning example with a cycle.

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

TABLE I: Formulation of the compute partitioning problem.

because there would be no path from $\mathtt{R}_2$ to $\mathtt{W}_1$ with a *single* backward edge.

B. Abstract Resource Mapping

The output of the imperative to dataflow lowering (§III-A) is a VUDFG that spatially pipelines the CFG in a streaming dataflow fashion. The abstract resource mapping phase maps each VU into a physical unit (PU) based on resource constraint. This section introduces two types of resource partitioning: partitioning of a large dataflow graph within a hyperblock across distributed CUs, and partitioning of a logical tensor across distributed MUs due to capacity or bandwidth limits.

Once all VUs in a dataflow graph can fit a PU, SARA attempts a *global merging* step that maps multiple VUs into the same PU to reduce resource fragmentation whenever possible. These merged VUs still operate independently when sharing the resources within a single PU.

1) Compute partitioning

The *compute-partitioning* phase breaks up VUs with large hyperblocks that do not fit in any PU. We formulate this as a graph partitioning problem shown in Table I. For a Plasticine PU, the number of stages in a PU restricts the number of operations in a hyperblock, the number of I/O ports limits the number of live-in/out variables, and the number of pipeline registers constrains the number of live variables between adjacent operations. Since the Plasticine network is specialized to handle efficient broadcasts, the in/out-degree of a partition counts the number of broadcast edges with unique sources across partitions. Furthermore, the partitioned subgraphs cannot form cycles; VUs wait for all input dependencies. As a result, cycles across VUs cause deadlock. However, cycles from LCD (such as accumulation) in the original DFG do not count; SARA initializes LCD edges with dummy data to break the cyclic dependency. Fig. 6 shows examples of valid and invalid partitioning. Solution 2 is better than Solution 1 because of less connectivity across partitions. Solution 3 is illegal due to the cycle between Partition 1 and 2. Fig. 7 shows an example with an LCD in the input graph. Solution 2 is invalid because unlike Solution 1, removing the LCD edge does not eliminate the cycle between partitions.

a) Retiming

Imbalanced data paths across partitions can cause pipeline stalls if long-lived paths are not sufficiently buffered [29]. To ensure full-throughput pipelining, SARA inserts retiming buffers that introduce new partitions, contributing to the cost in Table I's objective.

b) Global Merging

We formulate the global merging in Fig. 3 as a very similar problem as the partitioning problem, except working on the VUDFG dataflow graph instead of the dataflow graph within each VU. The merging process essentially assigns VUs into larger VUs that can still fit in a PU, which is a generalization of the partitioning problem with more complex constraints. We extend our algorithms for compute partitioning to handle merging across VUs.

Next, we present two algorithms to solve the graph partitioning problem described in Table I: a traversal-based algorithm, and a convex-optimization-based algorithm.

c) Traversal-based solution

To address the cycle constraint, the algorithm performs a topological sort of the dataflow graph. Starting from one end of the sorted list, the algorithm iteratively adds nodes into a partition until it reaches capacity, and then repeats with a new partition. The topological sort ignores the back edges of the LCD cycles during traversal and guarantees no new cycle is introduced. The resultant partitioning is a function of the traversal order. We evaluate both the depth-first search (DFS) and breadth-first search (BFS) algorithms with forward and backward dataflow orders. For DFS, we re-sort the remaining list each time when starting with a new partition.

d) Solver-based solution

We formulate the graph partitioning as a node-to-partition assignment problem solved using Mixed Integer Programming (MIP). Table III shows the detailed formulation. At a high level, we use a boolean matrix B to track the assignment, where $B_{i,j} = 1$ indicates node i is assigned to partition j. The values of B are selected by the solver and each node is restricted to a single partition assignment. Costs and objectives from Table I are formulated as a function of B, the DFG structure, and the hardware specification. Constraints such as the number and the type of operations supported by each type of PU are relatively simple to formulate with the solver's built-in functions. Input/Output arity constraints are substantially more complex due to the broadcast capability of the hardware,

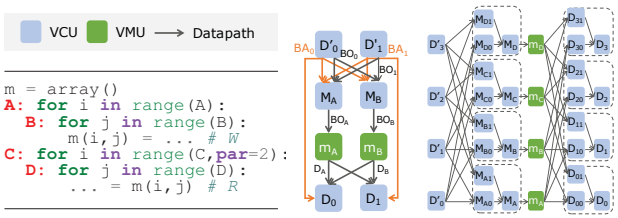
Name	Type	Description	Definition / Default
N	Constant, $\mathbb{Z}_{\geq 0}$	Number of nodes in the DFG	Input from the compiler
P	Constant, $\mathbb{Z}_{\geq 0}$	Number of partitions to consider	$\min(N, \text{hardware constraint})$, or from heuristic
$\mathcal{E}$	Constant, $\{i \to j\}$, $i, j \in [0, N)$	Directed edges of the DFG	Input from the compiler
c_O	Constant, $\mathbb{Z}_{\geq 0}$	Maximum output arity of a partition	HW Spec
c_I	Constant, $\mathbb{Z}_{\geq 0}$	Maximum input arity of a partition	HW Spec
b_d	Constant, $\mathbb{Z}_{\geq 0}$	Maximum input buffer depth	HW Spec
K	Constant, $\mathbb{R}_+$	Very large constant, used for constraint activation	$P \cdot N$
F	Constant, $\mathbb{B}^{N \times P}$	Feasibility matrix, where $F_{i,p}$ indicates whether node i can be assigned to partition p.	HW Spec
α_d	Hyperparameter, $\mathbb{R}_+$	Retime merging probability multiplier	$\frac{1}{\min\{c_O, c_I\}}$
B	Variable, $\mathbb{B}^{N \times P}$	Boolean matrix, where $B_{i,p} = 1$ indicates node i is assigned to partition p	Solver-initialized, or warm started by the traversal-based algorithm
d_n	Variable, $\mathbb{Z}_{\geq 0}^N$	Vector of node delays	Solver-initialized
d_p	Variable, $\mathbb{Z}_{\geq 0}^P$	Vector of partition delays	Solver-initialized
$\text{proj}_{\mathbf{B}}(v)$	$\mathbb{Z}_{\geq 0} \to \mathbb{B}$	Returns a constrained variable E that converts non-negative integer vector v into a boolean vector. Serves as an elementwise indicator in a convex formulation.	$\text{proj}_{\mathbf{B}}(v) \coloneqq \{E, E = 1 \text{ if } v \geq 1, \text{otherwise unconstrained}\}$
$\text{and}(a, b)$	$\mathbb{B} \times \mathbb{B} \to \mathbb{B}$	Logical AND in a convex formulation	$\text{and}(a, b) \coloneqq \max(a + b - 1, 0)$
$\text{any}(v)$	$\mathbb{B}^n \to \mathbb{B}$	Returns if any element in vector v is greater than 1	$\text{any}(v) \coloneqq \text{proj}_{\mathbf{B}}(\Sigma_k v_k)$
$\text{dest}(\text{i})$	$[0, N) \to \{[0, N)\}$	Returns the destination set of the source node i	$\text{dest}(i) \coloneqq \{j \mid \{i \to j\} \in \mathcal{E}\}$
$\text{dest}_{\text{P}}(i)$	$[0, N) \to \mathbb{B}^P$	Returns a boolean vector x, where $x_p = 1$ if node i has a destination in partition p, and i is not in p	$\text{dest}_{\text{P}}(i) \coloneqq \max\{\text{proj}_{\mathbf{B}}(\Sigma_{j \in \text{dest}(i)} B_{j,:}) - B_{i,:}, \vec{0}\}$
$\text{apart}(i, j)$	$[0, N) \times [0, N) \to \mathbb{B}$	Returns if node i and j are not in the same partition in a convex formulation	$\text{apart}(i, j) \coloneqq \max_p\{B_{i,p} - B_{j,p}, 0\}$
$\mathcal{C}$	$[[0, N) \to \mathbb{R}_+, \mathbb{R}_+, \mathbb{R}_+^n \to \mathbb{R}_+]$	Cost functions, resource limits, and aggregation functions of a list of resource constraints	HW Spec

TABLE II: Names and definitions used in the solver-based algorithms.

Type	Description	Expression
Cost Function (Objective)	# Allocated Partition	$\Sigma_{i \in [0,N)} \text{proj}_{\mathbf{B}}(\Sigma_p B_{i,p})$
	# Retiming Partition	$\alpha_d \Sigma_{i \to j \in \mathcal{E}} \text{proj}_{\mathbf{B}}(\max\{d_n(j) - d_n(i) - b_d, 0\})$
Partition Constraint	Partition Assignment[1]	$\forall i \in [0, N) : \ \Sigma_p B_{i,p} = 1$
	Input Arity Constraint	$\forall p \in [0, P) : \Sigma_{i \in [0,N)} \ \text{dest}_{\text{P}}(i)_p \leq c_I$
	Output Arity Constraint	$\forall p \in [0, P) : \Sigma_{i \in [0,N)} \ \text{and}(B_{i,p}, \text{any}(\text{dest}_{\text{P}}(i))) \leq c_O$
	Acyclic Constraint[2]	$\forall i \to j \in \mathcal{E} : \ d_n(i) + \text{apart}(i, j) \leq d_n(j)$
	Delay Consistency[3]	$\forall i \in [0, N) : d_n(i) \geq \underbrace{\max_p(d_p(p) + B_{i,p} \cdot K - K)}_{\text{Convex formulation for } d_p(p), \text{ where } B_{i,p} = 1}$, $d_n(i) \leq \underbrace{\min_p(d_p(p) + K - B_{i,p} \cdot K)}_{\text{Convex formulation for } d_p(p), \text{ where } B_{i,p} = 1}$
	Constant Validity	$\forall i \in [0, N) : \ d_n(i) \leq K, \ \forall i \in [0, P) : \ d_p(i) \leq K$
Merge Constraint	Feasibility Constraint	$\forall i \in [0, N), p \in [0, P) : \ B_{i,p} \leq F_{i,p}$
	Reducible Constraint[4]	$\forall p \in [0, P), \ \forall \ \underbrace{(c(\cdot), c_v, r(\cdot))}_{\text{Cost function, constraint value, reduction function}} \in \mathcal{C}: \ \underbrace{r_{i \in [0,N)}(c(i) \cdot B_{i,p})}_{\text{Example } r\text{: sum, max, union, etc.}} \leq c_v$

TABLE III: Solver formulation for compute partitioning in Disciplined Convex Programming ruleset [17], [35].

1. Enforce each node can only be assigned to a single partition. 2. Enforce no cycle across partitions. 3. Ensure $d_n(i)$ to match with $d_p(p)$ where p is the partition of node i. 4. Enforce the aggregated resource costs of the partition p do not exceed various resource limits.



(a) (b) (c)

Fig. 8: Memory partitioning for loop C unrolling by (b) 2x and (c) 4x. BA is not drawn in (c).

and are separately handled. To handle the acyclic constraint, we introduce the node and partition delay vectors, d_n and d_p, which represent schedules for each node and partition, respectively. In addition to enforcing the acyclic constraint, these vectors are also used for calculation where retiming is required and project the amount of introduced retiming VUs. Finally, we use the traversal-based solution to warm start the matrix B and the delay vectors to reduce the solver runtime.

2) Memory partitioning

The memory partitioner maps a logical tensor across distributed MUs either to address capacity limit or to scale on-chip memory bandwidth when the computation is parallelized. The latter is critical to scale performance with parallelization on an RDA. Fig. 8b shows how SARA handles the read access R when the user parallelizes loop C by 2x (SARA vectorizes across SIMD lanes when innermost loops are parallelized). As discussed in §III-A, SARA allocates a request $\texttt{VCU}_{D'}$ to generate access R's read requests and sends the response stream to $\texttt{VCU}_{D}$. When loop C is parallelized, SARA duplicates $\texttt{VCU}_{D}$ and $\texttt{VCU}_{D'}$ per unrolled lane. To increase the address bandwidth of $\texttt{VMU}_{m}$, SARA partitions the memory into two VMUs. Each request VCU generates a bank address (BA) stream that indicates which partition the request corresponds to, and a bank offset (BO) stream that indexes into the partition. On the request side, SARA allocates a merging VCU per memory partition to filter requests from all lanes to that partition. On the receiver side, SARA forwards BA stream to the receiver VCU, which is used to filter responses from all partitions to that lane. When highly parallelized, SARA creates hierarchical merging VCUs to scale network bandwidth, as shown in Fig. 8c.

This strategy sustains a linear performance scaling with parallelization at the cost of a worst-case quadratic resource increase. In the common cases, the crossbar between the VCUs and VMUs can be optimized away if the BA expression that selects the partition can be statically resolved.

C. Performance and Resource Optimizations

In this section, we discuss compiler optimizations that improve the runtime or reduce resource usage in RDAs.

a) Memory strength reduction (msr)

In addition to traditional strength reduction on arithmetic operations, SARA also replaces expensive on-chip memories with cheaper memories whenever possible. For example, SARA replaces a scratchpad with PU's input FIFO if all accessors of the scratchpad have constant addresses. This commonly happens when the address of the memory can be statically computed when a loop is fully unrolled.

b) Route-through elimination (rtelm)

For patterns where the content of a non-indexable memory ($M1$) is read and written to another memory ($M2$), SARA eliminates the intermediate access if the reader of $M1$ and the writer of $M2$ operate in lock-step.

c) Retiming with scratchpads (retime-m)

By default, SARA uses PU's input buffers for retiming. This option enables SARA to use scratchpads as deeper retiming buffers.

d) Crossbar datapath elimination (xbar-elm)

Although crossbars between VCUs and VMUs (§III-B2) are very expensive in the general case, the BA often can be statically resolved when using certain combinations of loop unrolling factors. In these cases, SARA reduces the crossbar between VCUs and VMUs into partial crossbars or point-to-point connections.

e) Read address duplication (dupra)

On RDAs, it is sometimes cheaper to perform redundant computation rather than broadcast results to multiple consumers with very different delays. For example, the BA forwarding path to $\texttt{VCU}_{D\#}$ in Fig. 8b must be sufficiently retimed to fully pipeline the entire DFG. The retiming cost is further increased in Fig. 8c. These optimizations duplicate the BA calculation in $\texttt{VCU}_{D\#}$ rather than forwarding them from $\texttt{VCU}_{D'\#}$.

IV. Evaluation

To show the scalability of SARA's approach, we first study how performance and allocated resources increase with parallelization of the program (§IV-A). Next, we quantify the benefits of our optimizations (§IV-B), and compare against the vanilla Plasticine compiler [41](§IV-C). Finally, we demonstrate the absolute performance of our target RDA by comparing to a Tesla V100 GPU (§IV-D).

a) Methodology

We configure Plasticine in a 20×20 layout, with a total of 420 PUs. For runtime, we use a cycle-accurate simulator integrated with Ramulator [25] to model $1\,\mathrm{TB/s}$ of HBM2 memory technology, comparable to our baseline GPU. For solver-based algorithms, we use a commercial optimization solver—Gurobi [18], with 10 threads.

An application that is too big to fit on an RDA must be segmented into smaller CFGs compiled by SARA independently. A runtime would execute these CFGs in time by reconfiguring the RDA. Automatically segmenting a large CFG is future work. Table IV shows a set of kernels with very different characteristics to mimic CFGs that can spatially fit on an RDA in real applications. *mlp*, *lstm*, *snet*, *kmeans*, *logreg*, *sgd*, and *gda* are compute-bound kernels that are also on-chip memory intensive. *bs* and *rp* are kernels with massive basic blocks. *pr* is a sparse kernel and *sort* is a memory-bound kernel with irregular access patterns.

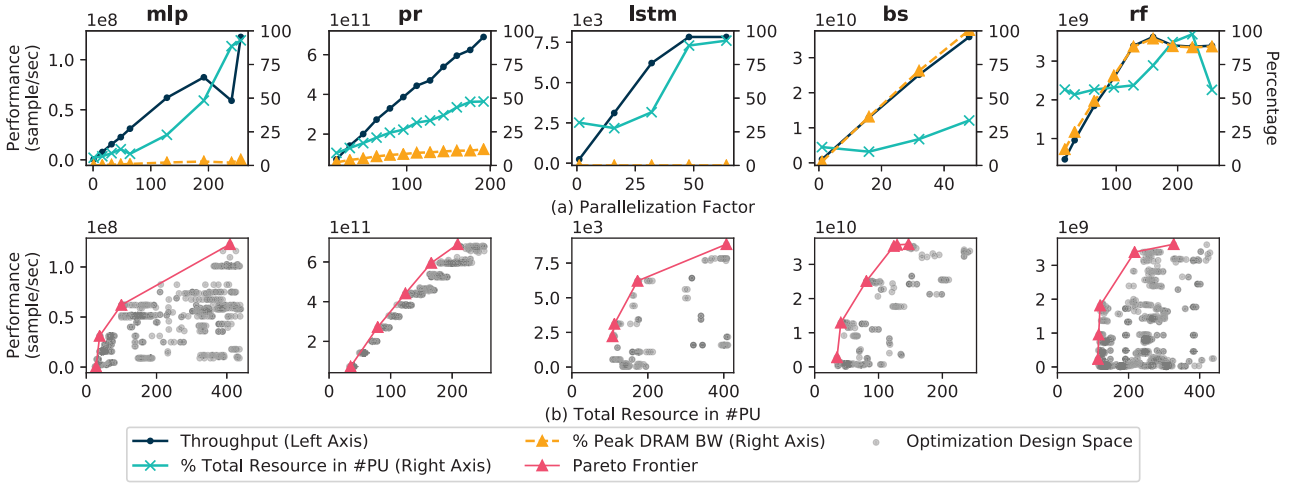


Fig. 9: (a) performance and resource metrics with increasing parallelization factors. (b) performance-resource tradeoff space.

Apps	Description	Data size/set
mlp	Multi-layer Perceptron Inference	batch-1 64×32×16×8
pr	PageRank	`delaunay_n20` [43]
bs	Black-Scholes	102400 records
rf	Random forest	10240 samples, 8 trees, 65 features max_split_per_node:64, tree_depth:40
lstm	LSTM RNN network [22]	H:512, time step:8
kmeans*	K-means clustering	50 iters, 1,536 points, 96 dims, K=20
gda*	Gaussian gradient descent	3,840,000 points, 96 dims
logreg*	Logistic regression	5 iters, 1,536 points, 384 dims
sgd*	Stochastic gradient descent	30 iters, 38,400 points, 768 dims
snet	SqueezeNet [23], a CNN network	batch-1, v1.0
sort	Merge sort	4194304 elements

TABLE IV: Benchmarks in all evaluations. *same size as [41]

(a) % resource and runtime improvement. (b) MLP case study.

Fig. 10: Optimization effectiveness.

We use single-batch *mlp* in our scalability study to show that SARA can achieve good scalability without trivial data-level parallelism. Due to the slow speed of our cycle-accurate simulator, we could not simulate very large ML models and scale Plasticine to the same size as the V100 GPU, which is 8x larger than our configuration accounting for the technology difference [46]. For comparison with the vanilla Plasticine compiler, we used the same Plasticine configuration as the original paper [41] tested on the same benchmarks.

A. Scalability of SARA

Fig. 9a shows the trend of performance and resource usage as we parallelize loops of a kernel. Starting with a fully-pipelined program, we incrementally parallelize the pipeline stage with the longest latency to balance the pipeline stages. We parallelize a kernel by unrolling multiple levels of nested loops, and the largest product of the nested parallelization factors (par factors) is shown on the x-axis. We present the best performant compiler configuration at each par factor. At a high level, Fig. 9a shows a perfect performance scaling (●), until running out of on-chip resources (×) for *mlp*, *pr*, and *lstm*, or saturating the HBM bandwidth (▲) for *bs* and *rf*. While *pr* only uses half of the PU, it exhausts all the on-chip memory PUs, which are half of the total PUs.

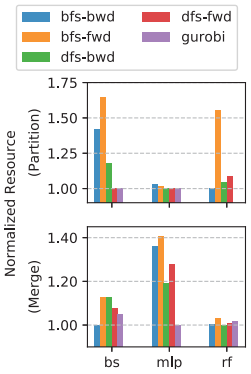
The case study in Fig. 10b shows that while performance linearly scales with increasing par factor *for a given configuration*, the resource may not monotonically increase, as power-of-two factors enable more optimizations. This explains the dip in Fig. 9a *mlp*; the best configuration for *mlp* did not fit the resource at par factor at 240, where we presented a less performant configuration. The best configuration does fit at 256, hence, restoring the linear scaling. The dip in *rf*'s resource (×) is due to the same reason. *rf*'s performance (●) saturates at par factor 128 due to DRAM BW (▲).

The linear performance scaling suggests that SARA can effectively utilize over 400 distributed PUs on Plasticine, and will extract more performance for on-chip resource-bound applications on larger Plasticine configurations.

Fig. 9b shows the performance-resource tradeoff space with different parallelization factors and optimizations. The pink curve outlines the Pareto frontier. This design space indicates a large variation in performance and resource usage due to compiler optimizations; these optimizations push the tradeoff curves leftward and upward, expanding the Pareto frontiers. The next section quantifies the effectiveness of individual optimizations.

B. Effectiveness and Impact of Compiler Optimizations

Fig. 10a shows the performance improvement and resource reduction enabled by different optimizations (§III-C). *merge* and *rtelm* give a consistent resource improvement for most applications. *dupra* and *msr* reduce the overhead caused by memory partitioning when applications are heavily parallelized, benefiting compute-bound applications like *mlp* and



(a) Resource Comparison

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

(b) Compile time for splitting

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

(c) Compile time for merging

Fig. 11: Partitioning and merging algorithm comparisons.

lstm. *retime* can have a substantial impact on the performance of the application at the cost of resource increase. Using scratchpad as a retiming buffer with *retime-m* can significantly reduce resource usage.

a) Traversal vs. solver-based algorithms

Fig. 11 shows the comparison between the traversal-based and solver-based solutions for compute partitioning and global merging. To speed up convergence, we configure a 15% optimality gap with Gurobi to stop the solver at a reasonable solution.

Fig. 11a shows the normalized number of PUs after partitioning and merging. We can see that Gurobi provides almost the best solution for all applications. The traversal-based algorithms can sometimes match the solver but can also be up to 1.7× worse in resource utilization than the best possible solution. Fig. 11b and Fig. 11c show the compile times using algorithms. The single-threaded traversal-based algorithm runs in minutes, significantly faster than the parallelized solver that can take hours to days. In summary, the solver solution provides a guaranteed close-to-optimal solution at the expense of compile time, while the traversal-based solution produces a decent solution, quickly. The former treats retiming and partitioning as a joint optimization, leading to a better overall solution. In practice, the traversal-based solution should be mostly sufficient and we can only invoke the expensive solver solution when the alternative yields a low resource utilization.

C. Comparison with the Vanilla Plasticine Compiler

We compare to the vanilla Plasticine compiler using a set of benchmarks from [41]. SARA improves upon PC in four aspects. First, SARA formalizes the token-based control flow with CMMC to enable an arbitrary number of accesses to a VMU, whereas PC only supports a single write and read access. Additionally, PC employs the hierarchical synchronization scheme (Fig. 2d) that introduces significant pipeline bubbles in deeply-nested control hierarchies, whereas SARA eliminates this overhead using a much more scalable P2P paradigm (Fig. 2c). Furthermore, SARA supports data-dependent control flow, such as branches, that PC does not support. The last improvement is the addition of memory partitioner (§III-B2) that partitions a logical memory across distributed memories (to scale capacity or bandwidth). Lack of a memory partitioner in PC limits the tiling size PC can support and prevents loops from being independently unrolled, limiting the application design space that PC can explore.

Benchmark	SARA				PC			Speedup
	PCU	PMU	BW	Par	PCU	PMU	Par	
kmeans	100%	80%	12%	128	11%	17%	16	14.4x
gda	77%	98%	15%	192	89%	88%	96	14.9x
logreg	100%	59%	100%	128	52%	69%	128	1.1x
sgd	77%	39%	88%	128	6%	9%	16	2.5x
Geo-mean								4.9x

TABLE V: Comparison with the Plasticine Compiler (PC)

App	Unit	Latency (ms)			Throughput (Unit/s)		
		SARA	GPU	Speedup	SARA	GPU	Speedup
snet	*kFrames*	49.1	70.1	1.4x	0.1 (1.0)	0.4	0.3x (2.5x)
lstm	*kSamples*	3.6	6.8	1.9x	8.8 (73.0)	4.7	1.9x (15.5x)
bs	*GOptions*	0.09	0.1	1.1x	88.9	80.0	1.1x
rf	*MSamples*	0.1	0.32	3.3x	1.0	0.3	3.3x
sort	*GElements*	0.6	2.1	3.4x	6.7	2.0	3.4x
pr	*MEdges*	128.3	829.4	6.5x	49.0	7.5	6.5x
Geo-mean				2.4x			1.9x (3.8x)

TABLE VI: Performance comparison of SARA against a Tesla V100 GPU with area-normalized speedup in parentheses.

Table V shows a comparison with benchmarks [41] using the same Plasticine configuration with a DDR3 DRAM. We focus on the compute-bound benchmarks as both PC and SARA can easily saturate DDR3 bandwidth ($49\,\mathrm{GB/s}$), resulting in a comparable performance for memory-bound and sparse benchmarks. We expect an up to 10× speedup when comparing on the HBM DRAM at $1\,\mathrm{TB/s}$, due to our retiming optimization that eliminates pipeline bubbles and merging optimization that enables larger par factors. *kmeans* and *gda* show a 14× speedup due to (a) larger par factors enabled by SARA and (b) 1.8× and 7.5× speedup from control overhead reduction with CMMC and other optimizations. *logreg* and *sgd* are less compute-bound; SARA saturates off-chip bandwidth before exhausting resources, resulting in a lower speedup. Overall, SARA significantly outperforms PC when mapping heavily compute-bound applications or targeting a fast off-chip memory, and we expect these benefits to grow with a larger chip size.

D. Comparison with a Tesla V100 GPU

Lastly, Table VI demonstrates the absolute performance SARA can achieve by comparing to an Nvidia Tesla V100 GPU. For GPU applications, we use TensorFlow [2] powered by cuDNN [10] for *snet* and *lstm*, a GPU graph library—GunRock [51]—for *pr*, CUDA libraries for *bs* and *sort*, and a hand-optimized CUDA implementation for *rf*.

SARA's lower throughput than GPU for *snet* is because our target Plasticine chip is 8.3× smaller than the baseline GPU. We show an area-normalized throughput for *compute-bound* applications, i.e. *snet* and *lstm*, in parentheses, where SARA is consistently faster than V100 when presented with the same amount of resources [46].

To utilize Plasticine's compute power for *bs* and *rf*, we fully streamline their program graphs in addition to exploiting data-level parallelism. Besides an efficient control mechanism that ensures minimum stalls in these deep pipelines, SARA further employs resource-reduction optimizations, enabling large par factors in our implementation that maxes out the HBM bandwidth. On the contrary, GPUs cannot exploit dataflow execution for *rf* to reduce memory accesses. The tree structures in *rf* introduce sparse memory accesses in GPU, leading to poor memory performance.

Similar to *rf*, *ms* also benefits from the dataflow execution model of Plasticine [50]. SARA's decentralized control scheme ensures full pipeline rates when reaching 100% utilization, resulting in a 3.4× speedup over the GPU.

Due to the limitation of GPU's SIMT abstraction, GunRock only parallelizes across the edge frontier of *pr*, providing insufficient parallelism to saturate GPU's compute capabilities in sparse graphs like the `delaunay_n20` dataset [43]. We exploit a flexible combination of edge- and node-level parallelism with SARA, leading to much more efficient usage of memory bandwidth.

V. Related Work

a) High-level Synthesis

Over the past few decades, there was rising interest in high-level synthesis (HLS) tools as alternatives to hardware description languages (HDLs) to build customized accelerators in different disciplines, especially in communication and signal processing domains [32]. HLS tools enable domain experts to program their algorithms in a software abstraction, often in C or C-variants, and output register transfer level (RTL) descriptions that can target an FPGA or generate an ASIC. There is a rich body of prior work on HLS techniques and many successful commercial HLS tools provided by FPGA vendors [1], [54].

At a high level, HLS bears some similarities to RDA compilation. Loop unrolling, pipelining, and memory partitioning are common techniques to target a reconfigurable architecture from a high-level imperative abstraction [26], [32], [52]. Similar to an RDA compiler, an HLS compiler also takes a specification of the hardware constraints and generates a wide design space with different design trade-offs, such as resource, performance, or energy efficiency.

Despite these similarities, there are fundamental differences between an HLS tool that generates RTL designs and an RDA compiler that programs an RDA.

While RDAs also have reconfigurable datapath, they are much more restrictive than FPGAs and cannot handle arbitrary RTL designs. HLS compilers often pre-build a set of parameterized RTL templates or soft processor cores that can be sized and optimized for a particular application [8], [40], [53]. Taking advantage of domain knowledge, domain-specific HLS compilers can generate highly customized datapaths for different application domains [13], [32]. RDAs, on the other hand, are reconfigured at a much coarser-grained level and cannot specialize execution engines for different applications. Hence, techniques like overlaying architectures are less applicable for RDA compilers.

HLS compilers often generate a hierarchical design to manage the design complexity using a divide-and-conquer approach [47]. Gluing these hierarchical modules often requires customized control logic at the RTL level that is either too inefficient or not supported on RDAs. Furthermore, HLS generated designs are not designed to be distributed. The control and datapaths cannot tolerate arbitrary network latency introduced by the RDA's global network.

b) CGRA Mapping

Prior works have proposed modulo scheduling techniques to map the dataflow graph *within* a hyperblock onto the accelerator array [34], [56]. Other works used integer linear programming, and hybrid optimization and heuristic techniques for CGRA PaR [37], [39]. These works correspond to phase two of Fig. 3, whereas we focus on the first phase—synthesizing a hierarchical dataflow graph that spatially pipelines the *entire* CFG. While we also optionally use convex optimization in SARA, we use it on graph partitioning rather than PaR.

Prior works, such as TRIPS [44] and WaveScalar [48], are hierarchical and distributed CGRAs with a hardware-managed shared memory system. Nonetheless, they are much smaller in scale, with tens of distributed units compared to hundreds of units in RDAs. One major distinction in the compilation strategy is how to map a program across these distributed units.

The TRIPS compiler uses inlining, loop unrolling, and control to data dependency conversion to maximize the size of a hyperblock [44]. The WaveScalar compiler breaks the CFG into larger blocks than a hyperblock with a single control entry, and uses speculation to further extend the ILP window [48]. In contrast, the ILP window for SARA is not bounded by any scope in a CFG, but rather by the resource limits of an RDA.

In these shared memory CGRAs, the compiler still exploits *thread-level* parallelism across distributed units. This scaling strategy would not scale due to the communication overhead in a shared memory system. On the other hand, SARA exploits hierarchical *pipelined* execution and nested loop-level parallelism *across* distributed units, where each hyperblock is a node in the coarse-grained dataflow graph. The pipelined execution is much more tolerant to network latency [55], and can effectively scale the performance of a compute-intensive kernel when presented with more on-chip resources.

Similar to Plasticine, the RAW microprocessor is a tiled architecture with a similar interconnect and distributed on-chip memories [49]. Their compiler Rawcc [27] also explicitly manages the scratchpads [6] and has a scheduler for tile placement [28]. Nonetheless, units in RAW are processors rather than statically configured pipelines like those in Plasticine, which are much more efficient and less flexible. As a result, our mapping strategies significantly differ. Rawcc relies on complicated memory disambiguation analysis to split independent instructions *within* a basic block into concurrent blocks mapped across units. In contrast, SARA spatially pipelines *different* hyperblocks across units and exploits *nested* loop-level parallelism to fully utilize the RDA's resources.

Additionally, Spatial expresses independent data structures as disjoint memories so SARA can easily detect independent memory accesses without complicated pointer analysis as in Rawcc. Lastly, in contrast to the per-request synchronization in RAW, SARA synchronizes memory accesses at a stream level, where each stream corresponds to a set of accessors from a single location in the program.

VI. Conclusion

RDAs are a promising class of reconfigurable accelerators delivering performance and resource density comparable to fixed-function accelerators, while capturing a large application space. However, to sustain these benefits with growing RDA sizes, the software stack must address the scalability challenges while providing a productive programmable interface for application developers.

We present SARA, a compiler that exploits multiple levels of dataflow and loop-level parallelism beyond a hyperblock to utilize a large-scale RDA using CMMC. Furthermore, our resource virtualization and composition scheme efficiently handles the constraints introduced by a hierarchical RDA, and maximizes resource utilization on an RDA with heterogeneous resources. Lastly, we show that SARA achieves a geo-mean speedup of 4.9× over the vanilla Plasticine compiler and 1.9× over a Tesla V100 GPU with a much smaller chip area.

References

[1] "Vivado high-level synthesis," , 2016.

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

[3] A. V. Aho, M. R. Garey, and J. D. Ullman, "The transitive reduction of a directed graph," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 131–137, 1972.

[4] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '83. New York, NY, USA: Association for Computing Machinery, 1983, p. 177–189.

[5] G. Ansaloni, P. Bonzini, and L. Pozzi, "Egra: A coarse grained reconfigurable architectural template," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 6, pp. 1062–1074, 2010.

[6] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, "Maps: a compiler-managed memory system for raw machines," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, 1999, pp. 4–15.

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

[8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36.

[9] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Seiociety, 2014, pp. 609–622.

[10] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, pp. 1–9, 2014.

[11] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[12] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[13] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran *et al.*, "Fast inference of deep neural networks in fpgas for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.

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

[15] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "Piperench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

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

[17] M. Grant, "Disciplined convex programming," PhD. Thesis, Stanford University, 2014.

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

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

[20] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Kressarray xplorer: a new cad environment to optimize reconfigurable datapath array architectures," in *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, 2000, pp. 163–168.

[21] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[23] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *ArXiv*, vol. abs/1602.07360, 2017.

[24] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 1–12, Jun. 2017.

[25] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan. 2016.

[26] 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*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 296–311.

[27] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a raw machine," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages*

and Operating Systems, ser. ASPLOS VIII. New York, NY, USA: Association for Computing Machinery, 1998, p. 46–57.

[28] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe, "Convergent scheduling," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 35. Washington, DC, USA: IEEE Computer Society Press, 2002, p. 111–122.

[29] Liang-Fang Chao and E. Hsing-Mean Sha, "Scheduling data-flow graphs via retiming and unfolding," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 12, pp. 1259–1267, 1997.

[30] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surv.*, vol. 52, no. 6, Oct. 2019.

[31] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," *SIGMICRO Newsl.*, vol. 23, no. 1–2, p. 45–54, Dec. 1992.

[32] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 18–25, 2009.

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

[34] B. Mei, S. Vernalde, D. Verkest, H. Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," 02 2003, pp. 296– 301.

[35] Y. Y. Michael Grant, Steven Boyd. (2019) Disciplined convex programming. Stanford University.

[36] C. Nicol. A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing. Wave Computing.

[37] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018.

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

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

[40] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 651–665.

[41] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 389–402.

[42] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," *SIGMICRO Newsl.*, vol. 12, no. 4, p. 183–198, Dec. 1981.

[43] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

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

[45] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE transactions on computers*, vol. 49, no. 5, pp. 465–481, 2000.

[46] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180nm to 7nm," *Integration*, vol. 58, pp. 74 – 81, 2017.

[47] Y. Sun, G. Wang, B. Yin, J. R. Cavallaro, and T. Ly, "Chapter 8 - high-level design tools for complex dsp applications," in *DSP for Embedded and Real-Time Systems*, R. Oshana, Ed. Oxford: Newnes, 2012, pp. 133–155.

[48] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 291–302.

[49] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[50] M. Vilim, A. Rucker, Y. Zhang, S. Liu, and K. Olukotun, "Gorgon: Accelerating machine learning from relational data," in *Proceedings of the 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE, 2020.

[51] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 11:1–11:12.

[52] 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*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 199–208.

[53] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 268–281.

[54] Xilinx, "The xilinx sdaccel development environment," , 2014.

[55] Y. Zhang, A. Rucker, M. Vilim, R. Prabhakar, W. Hwang, and K. Olukotun, "Scalable interconnects for reconfigurable spatial architectures," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 615–628.

[56] Z. Zhao, W. Sheng, Q. Wang, W. Yin, P. Ye, J. Li, and Z. Mao, "Towards higher performance and robust compilation for cgra modulo scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2201–2219, 2020.