Todo list

SARA: A Compiler for Scaling Reconfigurable Dataflow Accelerators

Paper #169

13 Pages Body, 14 Pages Total

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-efficient Reconfigurable Dataflow Accelerators (RDAs). However, scaling performance at large chip sizes demands new mechanisms of managing and utilizing these accelerators to saturate their compute throughputs.

To address this challenge, we present SARA, a compiler for efficient mapping of large-scale RDAs. SARA converts applications' complex control hierarchies, such as nested loops and branch conditions, into a streaming dataflow representation that can be efficiently executed by an RDA with distributed on-chip resources. SARA implements (a) a peer-to-peer (p2p) control paradigm inferred from an imperative programming style that minimizes synchronization overhead, and (b) a mapping strategy that decomposes the computation and memory in a program across a hierarchy of heterogeneous resources in a large-size RDA. Our evaluation shows that, by eliminating synchronization overhead with a distributed control-flow and improving resource utilization with a composable mapping strategy, SARA dramatically enhances RDAs' scaling rate on performance to the allocated resource. Over a mix of deep learning, graph processing, and streaming applications, SARA achieves a 4x speedup over a Tesla V100 GPU (with equal resource as the targeted RDA) by efficiently utilizing the accelerator.

1 Introduction

With the end of Dennard Scaling [7], the amount of performance one can extract from a CPU is reaching a limit. To provide general-purpose flexibility, CPU spends the majority of energy on overheads, including dynamic-instruction execution, branch prediction, and a cache hierarchy, and less than 20% of the energy on the actual computation [13]. Even worse, the power wall is limiting the entire multicore family to reach the doubled performance improvement per generation enabled by technology scaling in the past[8].

For this reason, many recent efforts are spent on leveraging application domain knowledge in hardware design to enable continued performance scaling while meeting the power budget[16]. Examples include widely adopted General-Purpose Graphics Processing Units (GPGPUs) in the deep-learning domain and machine learning (ML) accelerators, such as Tensor Processing Units [18] and EIE [14], providing orders of magnitude acceleration over a CPU. However,

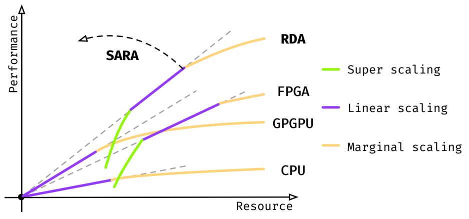


Figure 1. Scaling regimes for processor (CPU and GPGPU) and spatial architectures (FPGA and RDA).

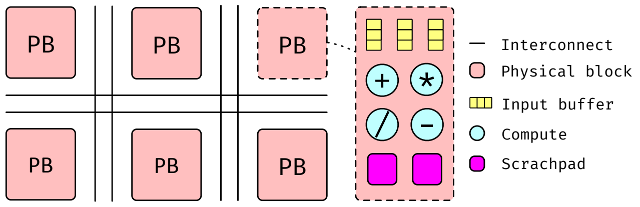


Figure 2. RDA High-Level Model

massive threads in GPU and the highly specialized datapath in ML accelerators often cause severe underutilization of the hardware due to variation in application and data characteristics[34].

Reconfigurable spatial architectures overcome this limitation by changing its datapath based on applications' needs. Applications are configured at the circuit-level without dynamic instruction fetching and decoding, hence improving energy-efficiency. In addition to instruction, data, and task-level parallelism explored by processor architectures, spatial architectures also explore instruction and task-level pipelining that further increase the compute throughput. Pipelining at various granularity enables spatial architectures to achieve high throughput on program phases without massive parallel workload, and allocate resource proportional to compute intensities of program phases. As the whole program is pipelined, performance is bounded by the bottleneck stage with the highest latency. Initially, the overall throughput can be doubled by only parallelizing the bottleneck stage without doubling the resource of the entire program, which gives a super scaling in performance to allocated resources, shown in Figure 1. After stage-balancing, programs enter the linear scaling regime with proportional performance to resource increase, similar to the best achievable scaling on processor architectures. Eventually, communication overheads cause sub-linearly scaling in all architectures. Many applications inherently contain imbalanced workloads in their program structures. For example, layers of a deep neural network vary highly in operational complexity. Spatial architectures can take advantage of imbalance workloads in a program to achieve high marginal performance improvement per resource increase in the super scaling regime. For applications, such as large image network like ResNet[15], RDAs tend to run out of resource before balancing the pipeline, which leaves the program entirely in the super scaling regime.

One example of a spatial architecture is Field Programmable Gate Arrays (FPGAs) that support fine-grain, bit-level reconfiguration with a soft logic fabric [21]. Although around for a long time, FPGAs are not broadly used on high-level applications due to their low-level programming interface and low resource density with high routing overhead.

Lately, Reconfigurable Dataflow Accelerators (RDAs) [26, 29] are emerging as a new class of spatial accelerators that retain the desired level of flexibility and energy efficiency without the fine-grained reconfigurability overhead. RDA provides high resource density and compute throughput with a hierarchical streaming network, operating at a fixed and high clock frequency at large chip sizes. Recent studies [29] have demonstrated a promising acceleration of dense, sparse, and streaming applications using RDAs.

Today, these RDAs are controlled using centralized-control schemes to manage and schedule the spatially distributed compute and memory resources. However, to accommodate applications with ever-increasing sizes (*e.g.*, large deep neural networks), architects are building ever-larger RDAs. At scale, the centralized schemes incur high synchronization overhead and communication hot spots. To achieve complex control schemes, such as a while convergence, a host often has to launch the RDA multiple times and materialize the on-chip states to DRAM, which further degrades performance. Moreover, as RDAs become larger, their coarse-grained hierarchal structure introduces fragmentation in resource allocation. Traditional mapping strategies—performing one-to-one allocation of resource tiles to program constructs—results in severe underutilization of resources, and heterogeneity in RDA resources further decreases mapping efficiency.

In this paper, we introduce SARA—a compiler for scaling performance and enhancing mapping efficiency for RDAs. We propose a distributed p2p control paradigm that minimizes synchronization overhead for RDAs at scale. We show that SARA can infer synchronization required for on-chip distributed execution from an application, written in a sequential, unpipelined programming abstraction. We present a mapping strategy efficiently decomposing a program over the distributed collection of compute and memory resources with global optimizations, which significantly reduces resource fragmentation for RDAs with heterogeneous resources (§2). The distributed control improves RDA's performance by extending the linear region in the performance-resource curve in Figure 1. Whereas, the mapping strategy and optimizations reduce the fragmentation and, hence, pushes the scaling curve left.

Using a recently published RDA–Plasticine [29]–as the target architecture, our evaluation shows that SARA improves Plasticine's scaling efficiency (§4.1). We qualitatively evaluate the impacts of individual optimizations on performance and resource (§4.2), and compare to a state-of-art Tesla V100 GPU (§4.3). We begin with a brief background on RDA architecture and the input of SARA (§2), before detailing the design of our compiler (§3).

Yaqi: quantify this

2 Background

In this section, we provide a background on the state-of-the-art RDAs (§2.1) that SARA targets by reading and transforming an Input Graph (§2.2) generated from a high-level programming language.

2.1 Reconfigurable Dataflow Accelerator (RDA)

RDAs are high-throughput, low-latency, and energy-efficient architecture providing flexible datapath and streaming dataflow execution. RDAs are designed as (multi-level) hierarchical architectures to provide better resource density in a scalable fashion. The hierarchical network allows RDA to maintain a fixed high clock frequency at a large scale in contrast to an FPGA. Figure 2 depicts an RDA with two-levels of hierarchy. At the outer-level, an RDA is configured as a pool of distributed physical blocks (PBs) that are interconnected via a global network (an ultra high-bandwidth all-to-all streaming network). Whereas, at the inner-level, each PB consists of compute and memory elements connected through a reconfigurable data path. Communication latency within PBs, between local compute and memory resources, is a single cycle (guaranteed); while access across PBs can take variable and potentially unpredictable amounts of latency due to DRAM access or dynamic network congestion.

Inside PBs, the compute can be implemented using a systolic array [4], DySER array [10], FPGA fabric, or SIMD pipeline [29]. The on-chip memory is often in the form of an SRAM or an eDRAM, behaving as an explicitly managed scratchpad. Unlike CPUs with cache-coherent memories, in RDAs, a software manages transfers between scratchpads and off-chip memories—mitigating the substantial performance and energy overhead due to cache coherency [13]. Other resources, within PBs, are input buffers (for streaming data from the network) and configurable counter (to keep track of local state). Furthermore, PBs can differ in the type and amount of resources they hold. For instance, there can be specialized PBs for generating DRAM requests that are less arithmetic-intensive than regular PBs.

2.2 Input Control- and Data-Flow Graph

The input to SARA is a hierarchical control- and dataflow graph that captures the computation, memory, and control structure of the program. The control hierarchy (Figure 3, left) comprises of control constructs, such as loops

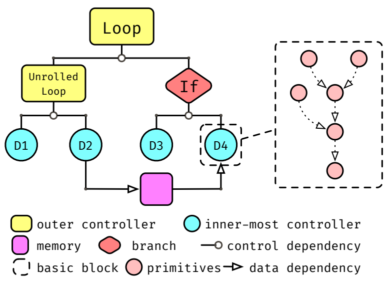


Figure 3. Input Control- and Data-Flow Graph for SARA

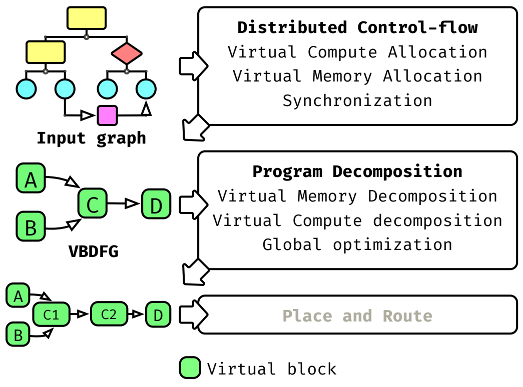


Figure 4. Compilation Phases.

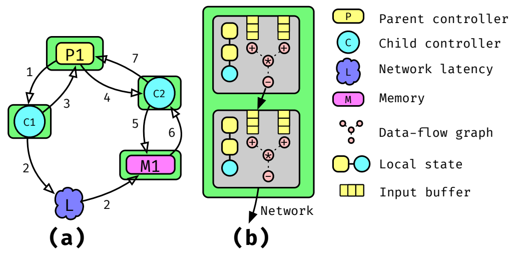


Figure 5. (a) Example of inconsistency in memory on a network with variable latency delay. (b) An actor that can execute data-flow graph of a leaf controller.

and branches. Each *inner most* controller contains a dataflow graph (Figure 3, right) belonging to a basic block with a single control entrance and exit. The input graph records intermediate memory accessed across controllers, carrying intermediate results of computations. The control hierarchy encodes how many times the computation is repeated as well as in which order the intermediate memories are accessed. A user or a high-level compiler determines which type and on/off-chip memory (such as DRAM, SRAM, or FIFO) the intermediate data reside in.

For on-chip SRAM, a user or a high-level compiler can *statically partition* the memory and input to SARA. Static memory partitioning (or static banking) [32], is a technique that statically transforms the data layout across memory partitions according to its access patterns in the program, such that parallel accesses do not have conflicts (accessing the same partition) at runtime. A user can explicitly specify a bank ID and an offset to indicate which partition and address within the partition to access, like in CUDA [24]. For a majority of access patterns, a high-level compiler[20] can automatically solve and inject the address transformations into the program.

3 Design of SARA

In this section, we describe a systematic approach to compile applications with a centralized control hierarchy into a distributed dataflow graph that can run on an RDA. Figure 4 shows our compilation flow. First, SARA takes the input graph and performs a *virtual block allocation.* A virtual block (VB) is our intermediate representation that eventually gets assigned to a PB. Communication across VB tolerates an arbitrary amount of network latency. The *VB allocation* phase allocates compute, memory, and necessary synchronization across VBs to produce the correct result. The output of the *VB allocation* is a VB dataflow graph (VBDFG) that can be executed on an RDA with infinite-sized PBs.

The next phase is the *PB allocation* phase that assigns each VB to execute on a PB. This step addresses VBs that do not satisfy PB constraints by decomposing them into multiple VBs. After all VB fits in at least one PB, we perform a global optimization that merges small VBs into a larger VB to reduce fragmentation in mapping. The output of the *PB allocation* phase assigns a PB type to each VB, which is input to a *placement and routing (PaR)* phase.

The PaR phase maps the VBDFG graph on the accelerator array. PaR on RDA is similar to PaR on an FPGA as studied in many previous works [33], which we do not discuss further in this paper. After PaR, SARA generates configuration for PBs which executes on the RDA.

In the following sections, §3.1 describes the major challenge addressed during *VB allocation*, which is converting the centralized control hierarchy into distributed controlflow. §3.2 details program-partitioning passes that decompose program over distributed resources. In §3.3, we talk about optional optimizations that either improves the runtime or reduces resource usages for an application.

3.1 Distributed Control Flow

In this section, we detail how SARA converts the controlinput graph (Figure 3) into a distributed VBDFG.

In a naïve approach, we can map each controller in the hierarchy into a VB (Figure 5). This strategy suffers from expensive network round-trip delays between the parent and child controllers. If the parent controller is an unrolled loop, the parent needs to synchronize with all child controllers, which creates an undesired communication hot spot. Figure 5(a) shows an example where synchronization *just* between parent and child controllers can produce an incorrect result due to unpredictable network latency.

The alternative approach explores a different way to execute the expected control schedule correctly. The minimum required synchronization to produce the correct result is to ensure that the computations access the intermediate results in a consistent matter as if the control schedule is strictly enforced. This can be achieved via p2p synchronizations *only* between computations that access a particular shared memory. The execution order of computations that access different memories does not need to be enforced, as they do not impact the program outcome. Therefore, as long as the compututation is executed with the expected number of iterations and the memories are updated consistently, there is no need for any extra synchronization. Next, we walk through how SARA achieves this in more concrete detail.

Yaqi: Add an example

Virtual Compute Allocation. As a start, SARA allocates workers to execute the dataflow graph within the *innermost* controllers of the input graph. We refer to our workers as *actors*. An actor can consume a set of input streams produced by other actors, perform operations, and send the output streams to other actors. The operations are from a basic block. Hence, they are *branch-free* and can be pipelined. The actor waits for all of its input dependencies before execution and stalls on downstream backpressure. A VB can contain one or multiple actors, as long as the total resource consumption of all actors satisfy the hardware constraint at the end. SARA assumes single cycle latency between actors within a single VB and variable latency for actors across VBs. SARA duplicates and distributes the outer controllers for each leaf controller to their corresponding actors, Figure 5(b). At this point, all actors have the local state to repeat their computation for expected iterations, completely asynchronously.

Virtual Memory Allocation. Next, SARA examines the intermediate results in the control hierarchy and allocates corresponding virtual memories. For each virtual memory, SARA generates synchronization between actors whose inner controllers read or write the memory within the input graph. SARA allocates a single-bit control token as an access grant to the shared memory and passes it between actors. The control token is no different from a regular boolean data-dependency. By controlling *where*, *how*, and *when* to pass the token, SARA is able to maintain a consistent update ordering between the pipelined and parallelized actors that access the shared memory.

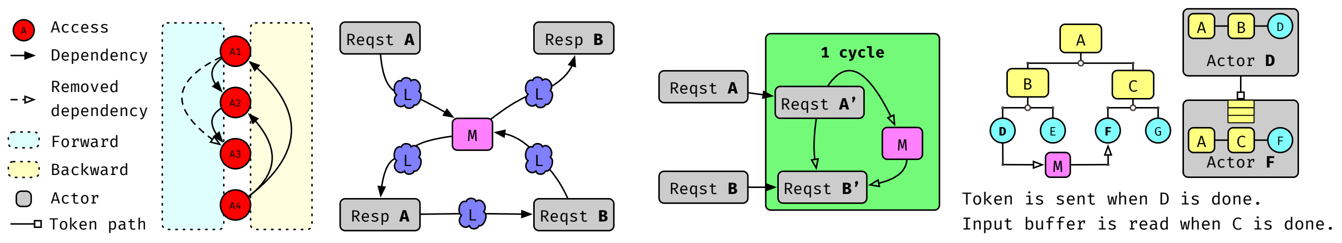
3.1.1 Synchronization

We refer to an access to the memory in the input graph as a *declared access*, as supposed to accesses executed at runtime. For example, multiple accesses across loop iterations are counted as a single declared access.

Where. SARA only allocate resource to synchronize actors if their declared accesses can potentially interfere. Whether two declared accesses interfere depends on the type of accesses, the type of the memory, and location of the accesses in the control hierarchy. For every declared access, SARA checks other accesses of the same memory appeared earlier in the program order for a possible forward dependency, and later in the program order for a possible loop-carried dependency (LCD). Two declared accesses A and B have no dependency if their least-common ancestor (LCA) controller executes only one of the children at anytime (from a branch), or all children in parallel (from an unrolled loop). The LCD exists between B to A, if B occurs later in the program order, and A and B are surrounded by a loop. To detect LCD, SARA checks if a loop exists among two accesses' LCA controller and LCA's ancestor controllers. For a dual-ported SRAM, all other accesses need to be synchronized to: share address ports for read-after-read (RAR) and write-after-write (WAW); enforce true data-dependency for read-after-write (RAW); and prevent the overriding of read data for write-after-read (WAR). If the memory is multi-buffered (by the user or high-level compiler), we do not need to synchronize for WAR [28] (the writer writes to a different buffer than the reader does). The DRAM interface permits concurrent read streams and, hence, RAR does not need to be synchronized.

How. For each intermediate memory, SARA builds a dependency graph for all its declared accesses (Figure 6(a)). Enforcing all dependencies in this graph may not be necessary as dependencies between A_1 and A_2, and A_2 and A_3 already capture the dependency between A_1 and A_3. Therefore, SARA performs a transitive reduction (TR) on the graph to keep the minimum number of dependency edges that preserve the same ordering [2]. Since TR on a cyclic graph is NP-hard, we perform TRs on the forward and backward LCD graphs, separately. Notice, dependencies between accesses touching different buffers of a multi-buffered memory is less rigid than accesses touching the same buffer. Therefore, we can only remove an edge if all dependencies on the equivalent path have a stronger or equivalent dependency strength than the strength of the removed edge.

To eliminate the round-trip overhead between the memory and the computation, SARA duplicates the local states and expressions required to generate the requests in a separate actor as the one that handles the responses. For write accesses, the memory provides an acknowledgment for each request received, used by SARA for synchronization. The request actor generates requests asynchronously as soon as its data-dependencies are cleared, pushing all requests to memory until back-pressured. To order a declared access A before a declared access B SARA creates a dummy dependency between the actor that accumulates the response of access A ($resp_A$) and the actor that generates requests for access B ($reqst_B$) (Figure 6(b)). To enforce LCD from access B to access A, SARA introduces a token from $resp_B$ to $reqst_A$, and initializes the token buffer (input buffer receiving the



(a) Access dependency. **(b)** Synchronization. **(c)** Single-cycle special case. **(d)** Token sending mechanism.

Figure 6. (a) Access dependency graph. (b) Synchronization of two accesses on the same memory. (c) Single-cycle special case. (d) Actors uses local states of controller hierarchy to determine when to send a token.

token) with one element to enable the execution of the first iteration. If the LCD is on a multi-buffered memory, the LCD token is initialized with the buffer depth number of elements to enable A for multiple iterations before blocked by access B.

These are general schemes we use on any types of memory (including DRAM and on-chip memories) with unpredictable access latency.

Special Case: Single-Cycle Access For a memory with guaranteed *single-cycle* access latency, such as registers and statically banked SRAMs that are guaranteed conflict-free, we can simplify the necessary synchronization (Figure 6(c)). Instead of synchronizing between $resp_A$ and $reqst_B$, we allocate two stateless actors $reqst'_A$ and $reqst'_B$ within the same VB as the accessed memory that forwards requests from $reqst_A$ and $reqst_B$, respectively. Next, we forward the token going from $reqst_A$ to $reqst_B$ to go through $reqst'_A$ to $reqst'_B$ instead, and configure the token buffer in $reqst'_B$ with the depth of one for serialized schedule and depth of M for multi-buffered schedule. We no longer need to insert the LCD token, as the stiff back pressure from the token buffer in $reqst'_B$ will enforce the expected behavior. This optimization only works if the sender and receiver of the token buffer are physically in a single VB where the memory is located. In this way, when $reqst'_B$ observes $reqst'_A$s token, $reqst'_B$ is guaranteed to observe the memory update from $reqst'_A$ because the memory also has single-cycle access latency.

Memory Localization We perform another specialization on non-indexable memories (registers or FIFOs), whose all accesses have no explicit read enables. Instead of treating them as shared memories, SARA duplicates and maps them to local input buffers in all receivers, no longer requiring tokens. The sender actor pushes to the network when the token is supposed to be sent, and the receiver dequeues one element from the input buffer when the token is supposed to be consumed. This dramatically reduces the synchronization complexity of non-indexable memory in the common case.

When. SARA configures the actors to generate the token using their local states at runtime. For FIFOs, the token is generated and consumed every cycle when the producer and receiver actors are active. For register, SRAM, and DRAM the program order expects that the producer and consumer writes and inspects the memory once per iteration of their LCA controller, respectively. Since the producer and receiver both have their outer controllers duplicated in their local state, they have independent views for one iteration of the LCA controller, which is when the controller in their ancestors (that is the immediate child of the LCA controller) is completed (Figure 6(d)). The *done* signals of these controllers are used to produce and consume the token in actors, independently.

3.1.2 Data-Dependent Control Flow

Using the synchronization discussed in §3.1.1, we can support control constructs that typically are not supported on most RDAs, such as branch and a while convergence loop. The controllers in the input graph can also have data dependencies, such as loop ranges. The dynamic-loop ranges are handled as data dependencies to actors with *memory localization* described in §3.1.1. The branch condition is also treated as a data-dependent enable signal of controllers under branch clauses. If the controller is disabled, it is considered *done* immediately. Output tokens depending on the *done* signal will be immediately sent out. For a memory written inside a branch statement and read outside the branch (with a branch miss), the writer actor immediately sends out the token to the receiver as soon as the branch condition is resolved. With a branch hit, the controller waits until its inner controller completes before raising the *done* signal. A similar scheme is used to implement the while loop, where the while condition is a data-dependency of stop signal of controller X. The producer of the while condition also consumes its own output as an LCD. The condition is then broadcast to all other actors under the same while loop. The *done* signal of the while loop is raised when the condition's data-dependency evaluates to true. At this point, actors accessing memory within the while loop will send the token to access actors outside of the while loop, and enable them to access the intermediate memory.

After all actors and shared resources are allocated and synchronized, we simply put each actor and shared resource into their own VBs. The actors with single-cycle special case (§3.1.1) must be put in the same VB as the shared memory.

3.2 Program Decomposition

The output of the previous step (§3.1) is a VBDFG that correctly executes the original program on an RDA without resource constraint. The *PB allocation* phase enforces and addresses constraint violations given a hardware specification of the RDA. In the end, each VB is assigned to a PB type used by PaR to map VBs onto the actual PBs.

In contrast to a combined VB allocation and assignment approach, where each VB is allocated for a specific type of PB, our approach enables maximum resource utilization with specialization in PBs, because we model what resource each VB consumes and each PB processes without tracking what types they are. For example, an application without DRAM access can use the specialized DRAM address generator PB to perform other computations.

To enforce resource constraints, SARA uses a VB-PB bipartite graph to keep track of potential valid assignments between the two. We refer to PBs with connection to a VB as the domain of the VB $dom(VB)$. Starting from a complete bipartite graph, a list of pruning steps, each considers a different type of on-chip resource and incrementally removes the edges that violate their resource constraints. The hard constraints restrict VBs to PBs to assigned resources, such as scratchpads and memory controller interfaces. The soft constraints, such as the number of operations in a PB, are "fixable" by decomposing the VB. The soft constraint pruners contain fallback partitioning transformations that attempt to fix VBs with empty domains. The compilation stops if a VB has an empty domain that cannot be fixed. The partitioning and pruning can be an iterative process. For example, the memory partitioning can introduce new computation that requires compute partitioning. After all VBs have at least one PB in their domain, SARA triggers a merging step that compacts the small VBs with slack room in their PBs into larger VB to improve PB utilization. Next, we perform a quick heuristic check on the bipartite graph to see if there exists a possible assignment with enough PBs, and provide feedback on the critical resources, otherwise. Finally, SARA assigns each VB to a PB type with a backtracking search on the pruned bipartite graph and feeds the VBDFG with assigned PB types to the PaR phase. §3.2.1 and §3.2.2 detail the two major partitioning on memory and compute. We have another partitioner encoding valid rule to decompose a BlackBox IP block available on the RDA.

3.2.1 Virtual Memory Decomposition

The memory pruner addresses VBs with virtual on-chip scratchpad memories exceeding the physical limits in capacity or number of banks in a PB. Memories in the input graph can have arbitrary size and number of virtual banks. The PBs, on the other hand, contains a small number of fix-sized 1-D scratchpad banks.

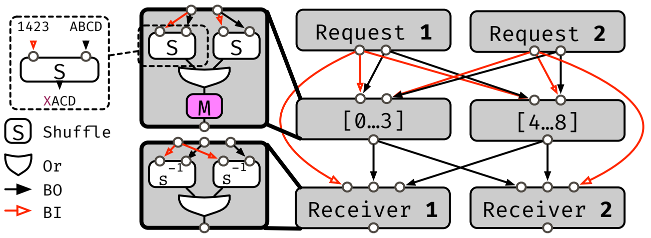


Figure 7. An example of splitting a memory to serve parallel requesters.

To partition the virtual memory, SARA shards the large virtual memory into multiple memory partitioned VBs, and assign each partition with a subset of the virtual banks. Each accessor provides a bank ID (BI) that selects which banks to access and a bank offset (BO) that specifies the address within the bank. BI and BO can be vectorized. SARA can use multiple banks within a PB to form a larger virtual bank. However, if a virtual bank in VB exceeds the total capacity of all physical banks within a PB, SARA further partitions the large virtual banks into multiple sub-banks, such that each sub-bank can fit into the aggregated capacity of a PB. To do so, SARA injects additional calculation to derive the sub-bank ID and sub-bank offset from the BO, and flattens the sub-bank ID with the previous BI to form the new BI and the new BO.

SARA then set ups the crossbar data path between the parallel producers, memory partitions, and parallel consumers. As discussed in §3.1.1, each accessor is split into a requester actor and receiver actor. For each memory partition, SARA uses an actor to merge the requests from all parallel requesters, Figure 7. The merge actor uses a special shuffle operator that shuffles the BO vector from the order in BI to the order aligned with banks in its partition. If a bank in the partition is not accessed in BI, the output of the shuffle is marked as invalid. We assume the capacity of each physical bank is much smaller than 2^{31}. So we use the first bit of the 32-bit BO to indicate invalid accesses (0 is invalid), which is also used to explicitly disabled access from the program. Next, the merger actor uses a tree of bit-wise OR operators to combine all bank-aligned BOs, and send the combined request vector to its partition. The requests can be trivially ORed because static banking (§2) guarantees that no two requesters access the same bank in the same cycle. Next, the memory partitions broadcast the respond to all receiver actors. A receiver takes response from all partitions, using the same shuffle operators to align each response back to the requested order in BI, and uses another OR tree to merge the response. The BI is forwarded from the requester to the receiver for the reverse shuffling.

Name	Type	Description	Definition / Default
$\mathcal{N}$	Constant	Enumeration of nodes to partition, numbered $\{n_i\}_i$	-
N	Constant, $\mathbb{Z}_{\geq 0}$	Number of operations to partition	$N = \lvert\mathcal{N}\rvert$
P	Constant, $\mathbb{Z}_{\geq 0}$	Number of partitions to consider	N, or from heuristic
$\mathcal{E}$	Constant, $\{n_i \longrightarrow n_j\}$	Directed edges representing dependence	-
B	Variable, $\{0,1\}^{N\times P}$	Boolean Partitioning Matrix	-
$\text{proj}_{\mathbf{B}}(\cdot)$	$\mathbb{Z}_{\geq 0} \longrightarrow \mathbb{B}$	Function to convert a positive integer into a boolean	Supplemental Materials
$\text{and}(\cdot,\cdot)$	$\{0,1\}\times\{0,1\} \longrightarrow \{0,1\}$	Boolean and of binary variables	Supplemental Materials
d_p	Variable, $\mathbb{Z}^P_{\geq 0}$	Vector of partition delays	-
d_n	Variable, $\mathbb{Z}^N_{\geq 0}$	Vector of node delays	-
$\text{dest}(n)$	$\mathcal{N} \longrightarrow \mathcal{P}(\mathcal{N})$	The set of nodes which depend on n	$\{n' \mid n' \in \mathcal{N}\ s.t.\ (n \longrightarrow n') \in \mathcal{E}\}$
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	$P \times N$
α_d	Hyperparameter, $\mathbb{R}_+$	Retime merging probability multiplier	$\frac{1}{\max\{c_o, c_i\}}$

Table 1. Names and definitions used in the solver-based partitioning.

Type	Description	Expression
Objective	Allocated Partitions	$\Sigma_i \text{proj}_{\mathbf{B}}(\Sigma_j B_{i,j})$
	Additional Retiming Partitions	$\alpha_d \Sigma_{n_i \to n_j \in \mathcal{E}} \text{proj}_{\mathbf{B}}(\max\{d_n(j) - d_n(i) - b_d, 0\})$
Constraint	Partition Assignment	$\forall n_i \in \mathcal{N} :\ \Sigma_j B_{i,j} = 1$
	Dependency Constraint	$\forall n_i \longrightarrow n_j \in \mathcal{E} :\ d_n(i) + 1[p_i \neq p_j] \leq d_n(j)$
	Output Arity Constraint	$\forall p \in [0, P) :\ \Sigma_{n_s \in \mathcal{N}} \text{and}(B_{s,p}, \text{proj}_{\mathbf{B}}(\max\{(\Sigma_{n_d \in \text{dest}(n_s)} B_{d,p}) - K \times B_{s,p}, 0\})) \leq c_o$
	Input Arity Constraint (vectorized)	$\Sigma_{n_i \in \mathcal{N}} \max\{\text{proj}_{\mathbf{B}}(\Sigma_{n_j \in \text{dest}(n_i)} B_{j,:}) - B_{i,:}, 0\} \leq c_i \times \vec{1}$
	Delay Consistency	$\forall n_i \in \mathcal{N} :\ d_n(i) \leq \min_j(d_p(j) + K - B_{i,j} \times K)$
		$\forall n_i \in \mathcal{N} :\ d_n(i) \geq \max_j(d_p(j) + B_{i,j} \times K - K)$
	Constant Validity	$\forall n_i \in \mathcal{N} :\ d_n(i) \leq K$
		$\forall i \in [0, P) :\ d_p(i) \leq K$

Table 2. Solver formulation for partitioning*.

Name	Type	Description	Definition / Default
C_r	$[\mathcal{N} \longrightarrow \mathbb{R}_+, \mathbb{R}_+, [\mathbb{R}_+] \longrightarrow \mathbb{R}_+]$	List of per-node values, limits, and reduction functions for reducible constraints	Supplemental Materials
F	$\{0,1\}^{N\times P}$	Feasibility matrix, whether a partition can support a node	HW Spec

Table 3. Table 1 extension for solver-based merging, which is a generalization of the partitioning problem.

Type	Description	Expression
Constraint	Feasibility Constraint	$\forall i, j \in [0, N) \times [0, P) :\ B_{i,j} \leq F_{i,j}$
	Reducible Constraints	$\forall j \in [0, P).\ \forall (c(\cdot), c_v, r(\cdot)) \in C :\ r([c(n_i) \times B_{i,j}]_{n_i \in \mathcal{N}}) \leq c_v$

Table 4. Table 2 extension for solver-based merging*. The Retiming Partition objective is not used for merging.

*Expressions are presented using the Disciplined Convex Programming ruleset [11, 22]. Explanations for selected expressions can be found in the supplemental material.

The alternative approach is to reverse the respond to access ordering within the memory partitions before sending them to the requester. This approach does not scale with network bandwidth, as the memory partitions need to send the receiver number of distinct outputs. (The number of receivers is a function of the parallelization factor.) As a result, the amount of output bandwidth at the memory partition limits how much the program can be parallelized, which causes underutilization of the accelerators. In our scheme, each partition sends a single broadcast to all receivers, which is efficiently handled by the network.

The request trees for memory partition and receiver can have high fan-in, which can be partitioned into a tree of VB during the compute partitioning phase in §3.2.2.

3.2.2 Virtual Compute Decomposition

Partitioning. The *compute-partitioning* phase addresses VBs using more compute than any PB can provide. If a VB contains multiple actors, SARA first attempts to move the actors

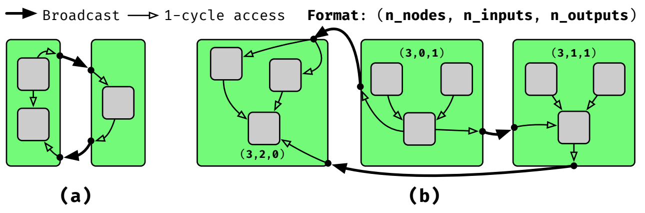


Figure 8. (a) An example of an illegal partition. (b) The cost of a partition.

into separate VBs. If a single actor exceeds the resource limit, SARA breaks down the large dataflow graph in the actor into multiple actors and put them in separate VBs. During partitioning, SARA maps each subgraph of the large dataflow graph into a new actor, mirrors the control states of the original actor, and streams live variables in between. The PB constraint includes the number of operations, input, and output ports available in the PBs. Because the global network is specialized to handle efficient broadcasts, the in- and out-degree constraint counts the number of broadcast edges, as supposed to edges between partitions (Figure 8(a)). In addition, the partitioned subgraphs cannot form *new* cycle between partitions that did not exists in the original VGDFG. Figure 8 (b) shows an illegal partitioning that fails the cycle constraint. This is because each actor is enabled atomically by all of its data-dependencies; cyclic dependencies between actors cause deadlock. The original dataflow graph, however, can contain cycles that correspond to LCDs. The back edge of the LCD is initialized with dummy data to enable execution in the first iteration.

Retiming. To pipeline all partitions at full-throughput, SARA must retime the imbalanced data paths between partitions with sufficient buffering. Retiming can introduce new VBs in addition to partitioned VBs. The objective of this phase is to minimize the number of partitions after retiming and minimizing the amount of connectivities between partitions.

The partitioner "fixes" the VB based on a single PB specification, albeit there are many potential PBs the decomposed VB can be mapped to. Currently, we use a heuristic to select a PB type from the $dom(VB)$ right before the compute pruning as a guiding constraint for partitioning. In the following sections, we present a traversal-based solution that provides a decent solution with fast compile time, and a convex optimization solver-based solution that provides an optimum solution with long turn-around time.

Traversal-based Solution To address the cycle constraint, we perform a topological sort of the dataflow graph. The topological traversal ignores the back-edge in the graph and produces a list of traversed nodes. The compiler starts from the beginning of the list, recursively adds nodes into a partition until the partition no longer satisfies the hardware constraint, and repeats the process with a new partition. This approach guarantees that no cycle is introduced with $O(V + E)$ complexity, where V and E are the numbers of vertices and edges. However, the outcome of the partitioning is a function of the traversal order, which does not guarantee an optimum solution, which we experienced with depth-first search (DFS) and breadth-first search (BFS) with forwarding and backward traversals. For DFS, we re-sort the remaining list each time we start with a new partition.

Solver-based Solution Table 2 gives our formulation of the partitioning problem. At a high-level, we use a boolean matrix B to keep track of the assignment of nodes in the dataflow graph to partitions. B has dimension of number of nodes to number of partitions, where$B[i, j] == 1$ indicates an assignment of node i to partition j. Each node is constrained to have a single partition assignment. The input and output arity constraints show the formulation of the PB I/O constraints. These are the two most challenging constraints as we need to identify broadcast edges across partitions. To address the cycle constraint, we introduce a delay vector d with size equal to the number of nodes. The delay vector encodes a schedule to execute each node. A node cannot execute earlier than its input dependencies and cannot be scheduled later than its output dependencies (Delay Consistency). The dependency constraint further limits nodes belonging to the same partition to have the same delay. This delay variable is also used to calculate where retiming is required and projection of the amount of retiming VBs introduced. The final object is just the sum of partitioned VBs and retiming VBs. To limit B to a small size, we use the traversal solution to determines the initial column size of B.

3.3 Optimizations

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

Memory strength reduction (msr). Like traditional strength reduction on arithmetics, SARA replaces expensive on-chip memories with cheaper memories whenever possible. For example, SARA replaces a scratchpad with constant address in all accesses to a un-indexable memory, such as a FIFO. This commonly happens when producer and consumer loops of the memory are fully unrolled.

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 read of M1 and the write of M2 operates in lock-step.

Retiming with scratchpad (retime-mem). By default, SARA uses PB input buffers for retiming purpose. This option enables SARA to use scratchpad memory for retiming that requires large buffer depth.

Crossbar datapath elimination (xbar-elm). Although, crossbars between accessors and memory partitions (§3.2.1) are very expensive in the general case, the BI sometimes can be statically resolvable with certain combinations of parallelization on memory accesses. When BI is a constant, SARA can use this information to intelligently assign virtual banks to partitions that reduce the crossbar data path to a partial or a point-to-point connection.

Read request duplication (dupra). During memory partitioning (§3.2.1), instead of forwarding BI from the requester to receiver, SARA can also duplicate the BI with local state on receiver side, which eliminates the unbalanced data path at the cost of extra computation.

Global Merging (merge). After all VBs satisfy the hardware constraint, we perform a global optimization to compact small VBs into larger VBs. Merging has very similar problem statement as compute partitioning (§3.2.2) except with more constraints. The traversal-based algorithm requires a reference cost for PB to check if the merged VB still satisfy the hardware constraint. At each step of merging, we take the union of the domains of VBs within the current partition, and intersect with the domain of the merging VB. The caveat is that even with a non-empty intersection, the bipartite graph might not have a possible assignment, as merged VB might fit in a larger PB with insufficient quantity. Therefore, we perform a heuristic checking on feasibility of the bipartite assignment at each step of merging. The solver-based solution combines partition assignment with PB type assignment as a joint problem. The output of merging gives both partition assignment as well as a PB type assignment, which eliminates the risk of un-mappable bipartite graph due to merging in the traversal-based solution.

4 Evaluation

In this section, we evaluate the effectiveness of our compiler, SARA, in scaling the performance and resource utilization of RDAs. We first study the impact of parallelization (*i.e.*, replicating compute and DRAMs) on applications' performance, which is measured in terms of throughput, on-chip resources (*i.e.*, physical blocks, PBs), and off-chip bandwidth, §4.1. Next, we look at how various compiler optimizations (§3.3) affect RDA's performance and resource utilization, as well as, the time it takes to generate the final configuration under different partitioning and merging schemes, §4.2. Finally, we compare the absolute performance of our target RDA against a Tesla V100 GPU for a mix of deep learning (DL), graph, and streaming applications, §4.3.

Test Methodology. For our testbed, we modify Spatial [20]—a high-level language for programming accelerators—to generate the input graph, discussed in §2. SARA reads and processes this graph, and produces an output for our target RDA, Plasticine [29]—an emerging high throughput, low-latency, and energy-efficient RDA. We configure Plasticine in a 20×20 configuration, and measure its runtime using a cycle-accurate simulator. To model DRAM, we further integrate an open-source DRAM modeling tool, called Ramulator [19], with our simulator, which adds support for an HBM2 DRAM technology, operating at 1 Tb/s. We also add support for a commercial optimization solver (Gurobi [12]) to evaluate the solver-based graph partitioning and merging algorithms.

For performance comparison, we use a Tesla V100 GPU with a die area of $815\,\text{mm}^2$ using a 12 nm technology process. The corresponding Plasticine configuration has an area footprint of $352\,\text{mm}^2$ on a 28 nm technology process, which has 9x less transistors than the V100 GPU. Due to speed limitations of our cycle-accurate simulation, we cannot simulate a Plasticine configuration with equivalent computing power. And, hence, we present a normalize speedup to die area for compute-bound applications. For writing applications, we use TensorFlow[1], compiled with cuDNN library[5], for SqueezeNet and LSTM; a GPU graph library, GunRock[31], for PageRank; a CUDA library for BlackScholes and sorting algorithms along with a hand-optimized implementation of a Random Forest in CUDA.

4.1 Scalability of RDAs

Unlike CPU-based architectures where the performance and resource utilization of a system *both* increase linearly with parallelization (*e.g.*, loop unrolling a program); in RDAs, the number of resources increase sub-linearly or quadratically—depending upon the degree of program imbalance—with parallelization. In other words, increase in resources results in super-scaler increase in performance of RDAs (Figure 1). Our experiments confirm this behavior and show that applications, when compiled with SARA, achieve this super-linear increase in performance with respect to resources (Figure 9, Bottom), while still maintaining a linear increase with respect to parallelization (Figure 9, Top).

Performance vs. Parallelization. We measure applications' throughput, on-chip resources, off-chip DRAM bandwidth, and compute activation rates at runtime, as we parallelize the bottleneck stage of an application (Figure 9, Top).

Without parallelization, SARA can still transform an application using resource partitioning and pipelining techniques, and achieve high compute throughput.For example, SARA partitions and pipelines the large loop body of BlackScholes (Figure 9, Top: **bs**) and all comparison trees in Random Forests (**rf**), while sustaining high throughput, in tens of million samples per seconds. On the other hand, with large compute bodies and complex graphs, SARA can only utilize 10–50% of resources without parallelization for **bs** and **rf** (as well as Long Short-Term Memory Recurrent, **lstm**, neural network).

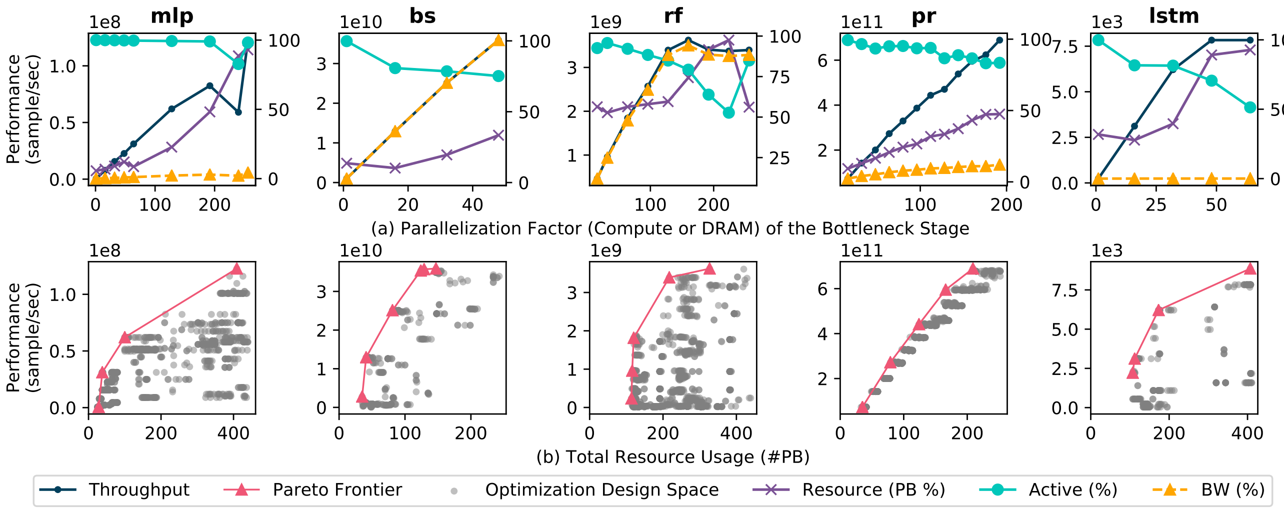


Figure 9. Comparing SARA's impact on the performance our target RDA, Plasticine, for different applications with varying parallelization factor (Top), and measuring the corresponding performance-resource tradeoff curve (Bottom).

With parallelization, the throughput compute-intensive applications (such as Multi-Layer Perceptron (**mlp**) and **lstm** scales linearly until all PBs are exhausted (reaching 100% uitilizaiton). However, the increase in resource utilization depends on the degree of program imbalance in an application. For example, consider an **mlp** application consisting of a single batch and three layers, with previous layer having twice as many operations as the next one. SARA can pipeline the layers and parallelize each layer into both inner and outer channels across PBs. Initially, there is a small increase in resource utilization as we increase parallelization, due to unbalanced layers; however, later, the resources increase quadratically due to all-to-all communication between parallelized layers. Furthermore, **mlp** achieves an activation rate—the percentage time a compute in the bottleneck stage is active at runtime—of about 100% at 100% resource usage, suggesting that **mlp** incurs minimum synchronization overhead with distributed control (§3.1). In the case of **lstm**, there is a loop-carried dependency between multiple time steps; parallelizing the inner product of a matrix-vector multiplication within each time step, increases the initiation interval of the carried dependency due to the increase in reduction-tree depth across PBs.

For memory-intensive applications, the DRAM bandwidth directly affects their performance. The throughput of **bs** and **rf** scales linearly until the DRAM bandwidth is saturated. The DRAM load stream—that fetches features from DRAM—limits **rf**'s throughput. We see a steady increase in its throughput as SARA parallelizes DRAM load (by loop unrolling) with minimal increase in resources—until we reach a parallelization factor of 150 (Figure 9, Top), after which the resources increase linearly. For PageRank (**pr**), we observe that the DRAM coalescing buffer captures the edge locality between nearest-neighbor nodes, when parallelizing nodes' processing across PBs and (vectorize) edge processing within PBs using pull-based graphs. Doing do, results in a dramatic reduction in off-chip memory access during sparse gather operations, thus providing a linear increase in throughput (*i.e.*, edge/sec) until our target RDA (Plasticine) runs out of PBs with memory controller interfaces.

We observe that for most applications, the resource utilization does not increase monotonically with parallelization; as for certain parallelization factors, SARA can statically resolve bank IDs and eliminate the crossbar communication path (*e.g.*, **mlp**, consumes less resources with a parallelization factor of 256 than 240).

Performance vs. Resource Usage. Figure 9 (Bottom) shows the throughput of our target RDA, normalized to the allocated on-chip resources. The gray dots indicate design points with varying combinations of compiler flags (*i.e.*, optimizations turned on or off), while the Pareto Frontier (in pink) shows the best combination of these flags. The Pareto Frontier confirms that RDAs indeed have a super-scaling relationship between performance and resources, as discussed in §2, for applications with significant program imbalance. **rf**, on the other hand, does not contain a heavily unbalanced program and, therefore, its performance can only scale linearly with resource usage at best. The large variation in throughput within the design space (*i.e.*, gray points) indicates that super scaling demands a collaborative effort in building new hardware and software optimizations.

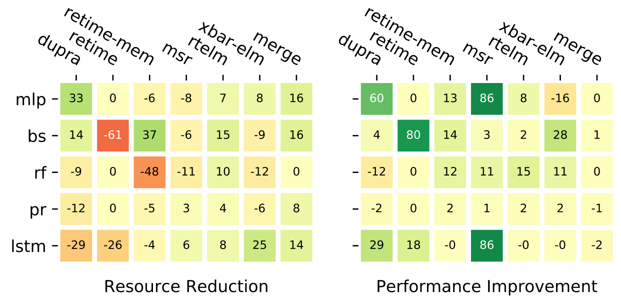


Figure 10. Maximum percentage improvement in resource usage and performance for different combinations of optimizations.

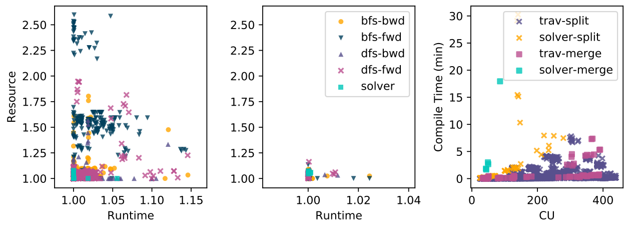


Figure 11. Normalized resource usage and runtime for traversal and solver-based partitioning and merging algorithms.

Benchmark (Unit)	**Compiler**	**Latency (ms)**	**Throughput (Unit/s)**
SqueezeNet (batch-1) *(kFrames)*	SARA	49.13	0.12 (1.1)
	TensorFlow	70.10	0.4
	Speedup	1.43x	0.31x (2.75x)
LSTM (batch-32) *(kSamples)*	SARA	3.61	8.8 (79.2)
	TensorFlow	6.81	4.7
	Speedup	1.89x	1.87x (16.85x)
PageRank *(MEdges)*	SARA	128.27	49
	GunRock	829.39	7.5
	Speedup	6.47x	6.5x
BlackScholes *(GOptions)*	SARA	0.09	88.88
	CUDA	0.10	80.02
	Speedup	1.11x	1.11x
Random Forest *(MSamples)*	SARA	0.10	1.04
	CUDA	0.32	0.32
	Speedup	3.25x	3.25x
Merge Sort *(GElements)*	SARA	0.63	6.65
	CUDA	2.14	1.96
	Speedup	3.4x	3.4x
Geo-mean Speedup		**2.44x**	**1.89x (3.93x)**

Table 5. Performance comparison of Plasticine with Tesla's V100 GPU (Normalized throughput to transistor count in parentheses).

4.2 Effectiveness of Compiler Optimizations

In this section, we evaluate the effectiveness of compiler optimizations (§3.3), as well as compare the traversal versus solver-based partitioning and merging algorithms.

Impact of optimizations. We evaluates the effectiveness of optimizations described in §3.3 on improving applications' resource usage and performance. As improvements due to compiler optimizations are non-additive, we pick the most feasible combination of these optimizations for each application and then enable and disable a given optimization to measure its impact on performance and resource usage (Figure 10). Each application has a large design space and as most optimizations are effective at higher levels of parallelization, we report a design point that has the maximum impact, either positively or negatively. *merge* and *rtelm* are optimizations that gives a consistent resource improvement for most applications. *dupra* and *msr* helps **mlp** and **lstm**, which are heavily partitioned during the memory-partitioning stage.

dupra significantly reduces unbalancing in the data path caused due to the large merge tree generated by memory partitioning. *retime* can have significant impact on performance of the application at the cost of increase in resource usage. Using scratchpad as a retiming buffer with *retime-mem* flag can significantly reduce resource usage.

Traversal versus solver-based algorithms. Figure 11 (a) and (b) show the resource usage and application runtime of traversal and solver-based partitioning and merging algorithms, normalized to the minimum resource and runtime for that application. A design point on the lower left corner (*i.e.*, at 1) indicates the best algorithm for the particular application.

SARA implements two graph structures that are commonly used for partitioning. (a) A dataflow graph consisting of large basic blocks, having long-lived variables that require retiming. (b) A tree structure used for merging memory requests; an ideal partitioning of these trees does not require extra retiming. We found that the backward BFS (bfs-bwd) produces the ideal partitioning for the balanced tree, whereas the backward DFS (dfs-bwd) produces fewer partitions in general. The backward traversal works better than the forward traversal because the graph tends to have smaller external out-degrees than in-degrees.

To speed up convergence, we use a 15% optimality gap that stops the solver at a reasonable solution. As we can see, the traversal based algorithm can sometimes achieve matching or even better runtime or performance than the solver. However, each traversal-based algorithm has adversarial cases where they can be up to 2.5x worse in resource than the best possible solution. The right most Figure 11 shows the compile time of both types of algorithms on different applications. In general, the solver runtime becomes quickly unscalable with a large amount of VBs. BlackScholes with less than 200 VBs took over 20 hours using 32 threads on an Intel Xeon Processor E7 server machine. In practice, we can only invoke the expensive solver only when the traversal solution is insufficient, which can be measured by the amount of slack resource in resulting partitions.

4.3 Comparison with a Tesla V100 GPU

Table 5 shows the runtime latency and throughput of Plasticine against a Tesla V100 GPU.

Neural Networks Comparing to a similar TensorFlow implemenation served on V100, SqueezeNet 1.0 implemented in SARA and served on Plasticine shows 1.5x latency speedup. It also shows 2.75x higher throughput. SqueezeNet benefits from the flexible degrees of parallelism SARA exploits. SARA can unroll any layer of the 6-level nested loops in the convolution layer and distribute them across PBs. As such, SqueezeNet can benefit from pipeline, channel, kernel, and task-level parallelism. In contrast, V100 only exploits data-level parallelism. Hence, SqueezeNet on V100 needs to accumulate a large batch size to fully utilize the accelerator's computation power and memory bandwidth. We also implemented a 1-layer, 8-step LSTM with 512 hidden units with SARA using the approach detailed in [34]. Our LSTM implementation achieves 1.8x latency speedup and 16.85x throughput improvement comparing to the same application served on V100.

BlackScholes and PageRank We ran BlackScholes implemented in CUDA and SARA till both enters steady-states. Both implementations reached similar performance due to HBM2 bandwidth limits. PageRank implemented in SARA Gunrock achieves 6.5x latency speedup and throughput improvement comparing to an implementation provided by Gunrock [31]. We found that Gunrock can only process the edge frontier set in parallel. Hence, Gunrock would not be able to fully utilize V100's compute power if the input graph is sparse. In contrast, SARA parallelizes both the edge and node processing with streaming DRAM data transfer. This feature allows us to exploit more degrees of parallelism within the PageRank algorithm. The input graph used in this application is denaulay_n20 [17].

RandomForest We created a synthesized tree model that contains 8 estimators and 128 splits per estimator. Implementation with SARA shows 3.2x in latency and throughput speedup comparing to one implemented in CUDA. We find that V100 drops performance when accessing the tree structure in each estimator due to bank conflicts. In contrast, SARA allocates the tree structures of the estimators spatially onto the PBs and statically schedules all the data accesses. As such, SARA manages to remove all the bank conflicts.

MergeSort We parallelize multiple multi-way merge tree that merges multiple DRAM streams. We use multiple iterations to incrementally multiple sorted DRAM partitions into a single sorted list, which achieves 3.4x normalized speedup in throughput than the fastest sort on GPU. Overall, MergeSort implemented in SARA provides 3.4x latency speedup and throughput improvement.

5 Related Work

Streaming Dataflow IRs Although many works claim to emit efficient and information-rich dataflow IRs for the downstream compilers, very few of them can capture the high-level parallel patterns and implementation details that are critical to RDA mappings. For example, TensorFlow [1] emits dataflow IR composed of tensor operations. However, its IR lacks information on the parallel patterns within these operations. In contrast, most of the streaming languages [9, 27, 30] are not able to extract nested loop-level parallelism from modern data-intensive applications. For example, StreamIt [9], a language tailored for streaming computing, also adopts distributed control as in SARA. However, it lacks the necessary language features to describe deeply and irregularly nested loops that are common in modern data-intensive applications.

Hardware Architectures Spatial reconfigurable accelerators (*e.g.*, Dyser [10] and Tartan [23]) have only one-level of hierarchy. Hence, such accelerators' performance can be bottlenecked by their limited interconnect bandwidth and power budget. Sparse Processing Unit (SPU) [6] can sustain higher interconnect bandwidth by introducing on-chip hierarchy; however, it lacks support for polyhedral memory banking [32], a pivotal optimization to achieve massive parallel accesses to on-chip memory. Plasticine [29] provides us with the desired architecture features; however, its compiler lacks the necessary components to support efficient streaming execution. Given that Plasticine resembles many key features of the RDA model, we target Plasticine with SARA.

Spatial Compilers Most previous works [3, 25] only consider allocating resources at the same level. SARA takes a more general assumption by co-allocating resources at multiple levels of an accelerator's hierarchy.

The Plasticine compiler [29] is similar to SARA that it also uses a token-based control protocol. However, it performs worse than SARA due to the following reasons. First, the Plasticine compiler allocates VBs for every level of Spatial's (a high-level language) control hierarchy. The communication between parent and child controllers lead to both communication hotspots around the parent, and bubbles before entering a steady-state of the loop iterations. Second, the Plasticine compiler assigns a single memory PB for each logical memory in the Spatial program. Hence, it could not handle the case where a logical memory exceeds the capacity or bank limits of the physical PBs. Third, the Plasticine compiler only supports polyhedral memory partitioning at the first dimension of the on-chip memory. Hence, its applicability to data-intensive applications with high-dimension tensor algebra is questionable. Last, compared to SARA's separate allocation and assignment phases described in §3.1 and §3.2, the Plasticine compiler allocates one VB for a specific type of PB and underutilizes resources within PBs.

6 Conclusion

Reconfigurable dataflow accelerators (RDAs) are a promising class of spatial accelerators, which deliver higher performance-to-resource efficiency than conventional process architectures while capturing a large application space. However, to sustain these benefits as RDAs get larger, the software stack must address the challenges of (a) distributed control / correctness and (b) efficient resource allocation.

We address these challenges by proposing a distributed asynchronous control scheme and a program decomposition method. We develop a compiler, SARA, that constructs a virtual block dataflow graph (VBDFG) from a program specification and generates a minimal set of peer-to-peer synchronizations, which allow fine-grained parallelization factors that would otherwise incur large communication overheads. Furthermore, operations on VBDFG are decomposed and assigned to a heterogenous collections of physical blocks (PBs). Lastly, we implement these techniques and through evaluations show that SARA achieves a speedup of 4x over a Tesla V100 GPU. We hope that the approach and implementation presented in this work will help scale modern RDAs add a steady pace..

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

[2] A. V. Aho, M. R. Garey, and J. D. Ullman. 1972. The Transitive Reduction of a Directed Graph. *SIAM J. Comput.* 1, 2 (1972), 131–137. https://doi.org/10.1137/0201008 arXiv:https://doi.org/10.1137/0201008

[3] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. 2004. Spatial Computation. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 14–26. https://doi.org/10.1145/1024393.1024396

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

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014), 1–9. arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[6] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. ACM, New York, NY, USA, 924–939. https://doi.org/10.1145/3352460.3358276

[7] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (Oct 1974), 256–268. https://doi.org/10.1109/JSSC.1974.1050511

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

[9] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. 2002. A Stream Compiler for Communication-exposed Architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 291–303. https://doi.org/10.1145/605397.605428

[10] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically Specialized Datapaths for Energy Efficient Computing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 503–514. https://doi.org/10.1109/HPCA.2011.5749755

[11] Michael Grant. 2014. *Disciplined Convex Programming*. PhD. Thesis. Stanford University.

[12] LLC Gurobi Optimization. 2019. Gurobi Optimizer Reference Manual. http://www.gurobi.com

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

[14] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 243–254. https://doi.org/10.1109/ISCA.2016.30

[15] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[16] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

[17] M. Holtgrewe, P. Sanders, and C. Schulz. 2010. Engineering a scalable high quality graph partitioner. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470485

[18] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross,

Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[19] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Comput. Archit. Lett.* 15, 1 (Jan. 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[20] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379

[21] Ian Kuon, Russell Tessier, and Julie Rose. 2007. FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electronic Design Automation* 2 (01 2007), 135–253. https://doi.org/10.1561/1000000005

[22] Yinyu Ye Michael Grant, Steven Boyd. 2019. *Disciplined Convex Programming.* Stanford University. https://web.stanford.edu/~boyd/papers/disc$_c$vx$_p$rog.html

[23] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. 2006. Tartan: Evaluating Spatial Computation for Whole Program Execution. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/1168857.1168878

[24] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53. https://doi.org/10.1145/1365490.1365500

[25] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. A General Constraint-centric Scheduling Framework for Spatial Architectures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 495–506. https://doi.org/10.1145/2491956.2462163

[26] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 142–153. https://doi.org/10.1145/2485922.2485935

[27] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 396–407. https://doi.org/10.1145/2594291.2594339

[28] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating Configurable Hardware from Parallel Patterns. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 651–665. https://doi.org/10.1145/2872362.2872415

[29] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 389–402. https://doi.org/10.1145/3079856.3080256

[30] Nemanja Trifunovic, Hristina Palikareva, Tobias Becker, and Georgi Gaydadjiev. 2017. Cloud Deployment and Management of Dataflow Engines. In *Proceedings of the 1st International Workshop on Next Generation of Cloud Architectures (CloudNG:17)*. ACM, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/3068126.3068795

[31] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. 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 (PPoPP '16)*. ACM, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/2851141.2851145

[32] Yuxin Wang, Peng Li, and Jason Cong. 2014. 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)*. ACM, New York, NY, USA, 199–208. https://doi.org/10.1145/2554688.2554780

[33] Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. 2019. Scalable Interconnects for Reconfigurable Spatial Architectures. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 615–628. https://doi.org/10.1145/3307650.3322249

[34] Tian Zhao, Yaqi Zhang, and Kunle Olukotun. 2019. Serving Recurrent Neural Networks Efficiently with a Spatial Accelerator. In *Proceedings of the 2 nd SysML Conference.*