MESA: Microarchitecture Extensions for Spatial Architecture Generation

Dong Kai Wang
University of Illinois at Urbana-Champaign
Champaign, IL, USA
dwang47@illinois.edu

Jiaqi Lou
University of Illinois at Urbana-Champaign
Champaign, IL, USA
jiaqil6@illinois.edu

Naiyin Jin
University of Illinois at Urbana-Champaign
Champaign, IL, USA
naiyinj2@illinois.edu

Edwin Mascarenhas
University of California, San Diego
La Jolla, CA, USA
emascare@ucsd.edu

Rohan Mahapatra
University of California, San Diego
La Jolla, CA, USA
rmahapat@ucsd.edu

Sean Kinzer
University of California, San Diego
La Jolla, CA, USA
skinzer@eng.ucsd.edu

Soroush Ghodrati
University of California, San Diego
La Jolla, CA, USA
soghodra@eng.ucsd.edu

Amir Yazdanbakhsh
Google Research, Brain Team
Mountain View, CA, USA
ayazdan@google.com

Hadi Esmaeilzadeh
University of California, San Diego
La Jolla, CA, USA
hadi@eng.ucsd.edu

Nam Sung Kim
University of Illinois at Urbana-Champaign
Champaign, IL, USA
nskim@illinois.edu

ABSTRACT

Modern heterogeneous CPUs incorporate hardware accelerators to enable domain-specialized execution and achieve improved efficiency. A well-known class among them, spatial accelerators, are designed with reconfigurability to accelerate a wide range of compute-heavy and data-parallel applications. Unlike CPU cores, however, they tend to require specialized compilers and software stacks, libraries, or languages to operate and cannot be utilized with ease by all applications. As a result, the accelerator's large pool of compute and memory resources sit wastefully idle when it is not explicitly programmed. Our goal is to dismantle this CPU-accelerator barrier by monitoring CPU threads for acceleration opportunities during execution and, if viable, dynamically reconfigure the accelerator to allow transparent offloading. We develop MESA (**M**icroarchitecture **E**xtensions for **S**patial **A**rchitecture Generation), a hardware block on the CPU that translates machine code to build an accelerator configuration specialized for the running program. While such a dynamic translation/reconfiguration approach is challenging, it has a key advantage over ahead-of-time compilers: access to runtime information, revealing not only dynamic dependencies but also performance characteristics. MESA maintains a real-time performance model of the program mapped on the accelerator in the form of a spatial dataflow graph with nodes weighted by operation latency and edges weighted by data transfer latency. Features of this dataflow graph are continuously updated with runtime information captured by performance counters, allowing a feedback loop of optimization, reconfiguration, and acceleration. This performance model allows MESA to identify the accelerator's critical paths and pinpoint its bottlenecks, upon which we implement in hardware a data-driven instruction mapping algorithm that locally minimizes latency. Backed by a synthesized RTL implementation, we evaluate the feasibility of our microarchitectural solution with different accelerator configurations. Across the Rodinia benchmarks, results demonstrate an average 1.3× speedup in performance and 1.8× gain in energy efficiency against a multicore CPU baseline.

CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; **Data flow architectures**.

KEYWORDS

spatial accelerator, binary translation, microarchitecture

ACM Reference Format:
Dong Kai Wang, Jiaqi Lou, Naiyin Jin, Edwin Mascarenhas, Rohan Mahapatra, Sean Kinzer, Soroush Ghodrati, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, and Nam Sung Kim. 2023. MESA: Microarchitecture Extensions for Spatial Architecture Generation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579371.3589084

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0095-8/23/06.
https://doi.org/10.1145/3579371.3589084

1 INTRODUCTION

Modern general-purpose processors suffer inherent inefficiencies due to von Neumann overheads [29]. Further worsened by the demise of Dennard scaling [16], this inefficiency has given rise to increasing adoption of specialized hardware for common application domains such as GPUs, DSPs, and more recently neural network and graph accelerators [1, 4, 9, 11, 14, 22, 24, 26, 27, 37, 38, 47, 58]. The rise of specialization yielded a similar trend towards heterogeneous CPUs with asymmetric core variants [19, 54, 60] and integrated on-chip accelerators [6, 62]. Hardware schedulers are also enhanced to handle the growing diversity in microarchitectures with the main mission of scheduling each thread to the most suitable compute and memory resources available. Within the same ISA domain, this is a well-explored problem; for example, Intel Thread Director [54] is an on-chip hardware controller that monitors thread execution activity to allow the scheduler to make better-informed decisions when scheduling between high-performance and energy-efficient x86 cores. The case for asymmetric multicores poses no threat to compatibility because the coexistence of multiple core microarchitectures supporting the same ISA is fully transparent to programmers, requiring neither source code modifications nor program recompilation. Extending this philosophy beyond CPU cores, the next step is to have a binary translation mechanism that retains transparency to software and programmer but adds accelerators as an execution target for CPU threads exhibiting characteristics amenable for acceleration.

We have two main **M**otivations for introducing such a binary translation mechanism. **(M1)** From a CPU perspective, pooling together accelerator resources as a shared scheduling target adds another dimension of specialized execution beyond microarchitecture variants. As modern CPUs are increasingly integrated with various types of on-chip accelerators, it is unlikely that most are heavily utilized by different applications. Hence, idle accelerator resources can be repurposed when they are not used conventionally. Furthermore, executing a thread on the accelerator can partially eliminate the von Neumann overheads present in CPU cores thereby improving energy efficiency. **(M2)** Spatial accelerators have a longstanding issue with ease of use and programming as each line of work tends to use its own suite of specialized compilers, libraries, languages, synthesis tools, or software stacks and there is no universal standard [3, 10, 12, 13, 17, 34, 35, 42, 43, 43–45, 50, 52, 59, 65]. Thus, from an accelerator perspective, dynamic binary translation (DBT) frees the hardware from total reliance on compiler and programmer involvement, eliminating a major barrier to adoption.

As a novel approach to this challenge, we propose MESA, a hardware controller on the CPU that performs three **F**unctions: **(F1)** monitoring program execution on CPU cores to assess viability for acceleration, **(F2)** translating the program binary to latency-weighted dataflow graphs (DFGs) and mapping them to a configurable spatial accelerator, and **(F3)** iteratively optimizing and reconfiguring the accelerator. As shown in Figure 1, MESA provides an alternative, low-cost, and transparent method of utilizing existing accelerator hardware. Since it operates at runtime, MESA uses a key feature not employed by traditional ahead-of-time compiler methods: using performance statistics gathered by activity counters on the CPU and accelerator to build a performance model

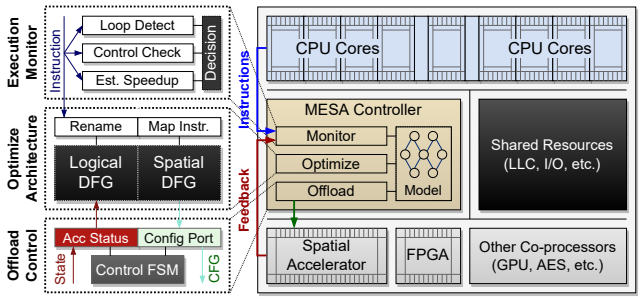


Figure 1: MESA is an on-chip hardware controller that allows certain CPU threads to be executed on spatial accelerators. It monitors CPU activity for acceleration opportunities, and, if viable, configures the spatial accelerator for the running program to allow dynamic offloading.

of the target code region. With the model and statistics, we propose a *data-driven* instruction mapping algorithm that can iteratively optimize the accelerator configuration based on runtime feedback.

To implement this feature, we introduce two hardware data structures: the *Logical DFG (LDFG)* which stores a linear view of the graph (indexed in program order) to reveal control and register data dependencies between instructions and the *Spatial DFG (SDFG)* which stores a planar view of the dataflow graph (indexed by position, out-of-order) exposing its instruction-level parallelism. These two structures represent the same graph stored in different formats; the LDFG, being linear, is used to maintain instruction ordering, and the SDFG, being planar, is used to configure the spatial accelerator. Additionally, the DFGs are weighted by measured latencies: nodes representing operations are weighted by their execution latency (cycles from inputs ready to outputs produced), and edges representing connections are weighted by their data transfer latency (cycles from parent's output to child's input). This weighted DFG is used by MESA as a dynamic performance model based on runtime feedback to estimate overall acceleration latency per iteration and rapidly identify the critical path and pinpoint nodes or edges that are sources of bottleneck. Using our performance model, we introduce a data-driven, locally latency-minimizing, and generally backend-agnostic hardware algorithm to map program instructions to spatial accelerators. Hence, unlike prior works in DBT [34, 65], MESA makes extensive use of runtime performance profiling to model and re-optimize its accelerator configuration.

Our results are backed by a detailed hardware implementation in SystemVerilog that supports the RISC-V (`RV32IMF` and `RV64I`) ISA tested on different configurations of a custom spatial accelerator. Evaluated using RTL simulation, MESA targeting spatial accelerators with 128 - 512 PEs achieves 1.33× – 1.81× speedup in performance, and 1.86× – 1.92× gain in energy efficiency over a multicore out-of-order CPU baseline while the MESA controller itself uses less than 10% of the area of a single core.

The main contributions of our work are as follows:

- **Data-driven Mapping Algorithm:** MESA builds and maintains a DFG-based architecture model based on measured performance data. With this model, we propose a greedy spatial mapping algorithm that locally minimizes expected instruction latency.

- **Iterative Optimization:** MESA uses runtime information continuously gathered from performance counters on the accelerator as inputs to iteratively optimize its spatial architecture and perform reconfiguration.
- **Iteration-level Parallelism:** For loops that are explicitly identified to be parallelizable (using OpenMP pragmas), MESA can additionally apply iteration-level parallel optimizations such as pipelining and unrolling.
- **Open Source Hardware Platform:** We design an RTL-level implementation of MESA that is open-sourced. This design can be synthesized or used for performance simulation; it can be interfaced with existing RISC-V cores [70].

2 BACKGROUND AND RELATED WORK

This section is a qualitative comparison of related work, we leave quantitative comparisons to Section 6.

CGRAs and Reconfigurable Accelerators. CGRAs and reconfigurable spatial architectures (RSAs) have seen a resurgence in academic research in recent years [3, 10, 12, 13, 17, 21, 35, 36, 42, 43, 43–45, 50, 52, 57, 59, 63]. Examples such as Stream-Dataflow [47], Plasticine [53], SARA [69], TRIPS [55], and DySER [18] also use spatial dataflow techniques to accelerate general applications. The main difference between MESA and these works is that MESA offers a transparent method to program the accelerator dynamically in hardware, which can complement ahead-of-time compiler-based methods. This transparency does come at extra cost in additional hardware logic and control limitations, but in turn, reveals runtime information that can be used for iterative optimization.

Dynamic Binary Translation. We discuss the differences in approach between MESA and earlier works in DBT. DynaSpAM [34] introduces microarchitectural additions to dynamically map program traces at runtime to a fixed feedforward CGRA on the CPU. Unlike DynaSpAM, MESA can target 2D spatial accelerators not restricted to a particular interconnect type. This is partly why we maintain an internal DFG model which adds complexity but can support different backend configurations and custom interconnects as long as point-to-point latency can be modeled. DORA [65] and [67] use a helper core to monitor execution and perform dynamic translation. MESA, on the other hand, takes the hardware implementation approach and uses a latency-driven algorithm for spatial mapping. While these works also exploit runtime information, MESA makes extensive use of per-instruction latency profiling for its mapping algorithm. Implemented in software, DORA has the advantage of flexibility but in turn, incurs much longer configuration time on the order of milliseconds compared to MESA's sub-microsecond time range. In this regard, DORA is more similar to a traditional compiler but executed alongside the CPU, whereas MESA's hardware approach is more restrictive but lightweight. Our goal is not to perfect the accelerator on the first configuration; we opt instead to continuously iterate to close in on the optimum. Furthermore, MESA differs in scope from both works in that it targets large spatial accelerators that are not intended to be integrated directly within the CPU pipeline. To saturate these arrays, MESA employs additional iteration-level optimizations when the program's parallel characteristics are known.

Microarchitecture Techniques. Various works in microarchitecture have proposed techniques to enhance the out-of-order backend. For example, methods of extended data forwarding between functional units [30, 49] or more elaborate designs such as CRIB [20] can completely bypass the register file; follow-up works like Revolver [23] further optimize loop execution. These techniques are confined to optimizing conventional microarchitectures and are limited in scope in comparison to MESA' goals of configuring full spatial backends. Other related works on dataflow processors such as DiAG [64] introduced drastic alternatives to the out-of-order approach through full-forwarding linear dataflow backends. However, DiAG is fundamentally a standalone CPU microarchitecture that is rigid in hardware whereas MESA is designed to configure spatial accelerators. Post-fabrication microarchitecture [31] bears some similarity to MESA in that it also attempts to dynamically construct hardware to optimize the running application. However, they primarily target the CPU's frontend to improve its speculative engines to adapt to applications with complex control structures, which are a different class of applications that MESA targets.

3 MESA MODEL AND ALGORITHM

At a high level, MESA maps an instruction sequence from the CPU to a spatial accelerator. This conversion is nontrivial because an ordered linear sequence of operations must be mapped to a planar grid of functional units and memory elements. Furthermore, since we propose MESA as a generalizable solution portable to different accelerator variants, little assumption is made on the organization of the target spatial accelerator. To facilitate this traversal from machine code to configuration bitstream, we introduce a middleground architecture abstraction layer in the form of a DFG equipped with some additional features. This DFG model serves two main **P**urposes: **(P1)** The DFG abstraction provides a data structure that can be easily manipulated such that the instruction mapping algorithm does not have to deal with underlying hardware directly. Our method assures that changes to the DFG are reflected in hardware during the eventual configuration step. **(P2)** The DFG allows performance modeling using measured delays at functional units and load-store units reported by the accelerator and/or CPU core. Armed with both a functional and performance model, we introduce a greedy hardware algorithm that locally minimizes the expected latency of each instruction.

This section describes MESA's DFG model and mapping algorithm; code region detection and memory/control handling is detailed in the next section. We introduce two data structures to represent the DFG in different formats: the *Logical Dataflow Graph (LDFG)* is indexed in program order (analogous to the CPU's reorder buffer) and provides a logical view of the DFG revealing inter-instruction data dependencies, and the *Spatial Dataflow Graph (SDFG)* is indexed by 2D coordinates and lays out a planar (spatially mapped) view of the DFG that represents how instructions would be assigned to the accelerator. Building an optimal SDFG from the LDFG is the main goal of the instruction mapping step, as the two structures are eventually used to program the accelerator. When a code region suitable for acceleration is detected, MESA performs three **T**asks:

(T1) Instructions → *Logical DFG* (Encode): MESA builds the LDFG from the code region executing on the CPU core to unravel its structure and dependencies.
(T2) *Logical DFG* → *Spatial DFG* (Optimize): Using the LDFG and captured performance data as input, we implement a hardware mapping algorithm that generates the SDFG by locally minimizing each instruction's expected latency.
(T3) *Spatial DFG* → Configuration (Decode): MESA maps the optimized SDFG to a programmable hardware backend with processing elements and load/store entries after which computation can be fully offloaded from the CPU.

3.1 DFG Background and Model

MESA models instruction dependencies using the standard dataflow compute model [28] with some additions. The DFG is a directed graph where instructions are represented by nodes, and dependencies between instructions are represented by edges. We assume that each instruction `i` ∈ {`i1`, `i2`, ...} has up to two predecessor instructions (source registers) `s1`, `s2` whose outputs are its inputs. For example, if instruction `i2` has a dependency on `i1` (i.e., `i2.s1 = i1`), then the DFG will have an edge (`i1`, `i2`) to denote that the output of instruction `i1` is used as input for `i2`. We assign the weight of each node to be the average estimated or measured latency of the node's operation (cycles elapsed from inputs available to outputs produced), and we assign the weight of each edge as the average latency of data transfer (cycles elapsed from the output of source node to the input of destination node). For convenience, we introduce the following notations:

L_i : cycle of completion of instruction `i`.
$L_{(i,j)}$: average latency of data transfer from `i` to `j`.
$L_{i.\text{op}}$: average latency of the operation of instruction `i`.

Under the dataflow model, an instruction can begin execution as soon as its inputs are available, regardless of original program order. We define the latency L_i of an instruction `i` as the number of cycles elapsed since the start of execution to the instruction's completion (i.e., the cycle at which the instruction produces its output). This latency is given by the cycle at which its predecessors' data arrive (A_{s1}, A_{s2}) added by the latency of the instruction's operation $(L_{i.\text{op}})$:

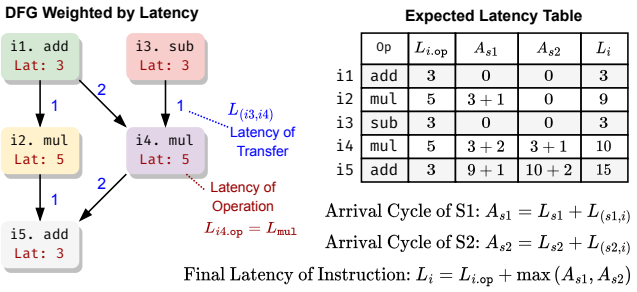
$$L_i = L_{i.\text{op}} + \max\left(A_{s1}, A_{s2}\right) \tag{1}$$

where $\max\left(A_{s1}, A_{s2}\right)$ gives the cycle the last input arrives at, since the operation cannot begin until all inputs are available. The cycle of data arrival A_{s1} can then be expanded as the latency of dependent instruction L_{s1} added by the latency of transfer from that instruction to the current $L_{(s1,i)}$. Thus, we can expand Equation 1 as follows.

$$L_i = L_{i.\text{op}} + \max\Big(\underbrace{L_{s1} + L_{(s1,i)}}_{\text{Arrival cycle of S1.}}, \underbrace{L_{s2} + L_{(s2,i)}}_{\text{Arrival cycle of S2.}}\Big) \tag{2}$$

Finally, the latency of an instruction sequence is the latency that all instructions are complete; this is given by the largest instruction latency: $\max\{L_{\texttt{i1}}, L_{\texttt{i2}}, \ldots\}$.

From the view of the DFG, the weight of a path is the sum of weights of nodes (operations) and edges (transfers) traversed; thus, the instruction latency L_i is given by the path with largest weight (critical path) that ends at i. Figure 2 shows an example



	Op	$L_{i.\text{op}}$	A_{s1}	A_{s2}	L_i
i1	add	3	0	0	3
i2	mul	5	3 + 1	0	9
i3	sub	3	0	0	3
i4	mul	5	3 + 2	3 + 1	10
i5	add	3	9 + 1	10 + 2	15

Figure 2: Example DFG with five nodes (instructions). The latency model is built using operation and data transfer latencies.

DFG with five instructions. In this example, we assume initial inputs are already available from registers and that the latency of addition/subtraction is fixed at 3 cycles and multiplication takes 5 cycles. The transfer latency between two nodes is modeled as the Manhattan distance between them, i.e., a single cycle for immediate neighbors and two cycles along the diagonal. On the right side of Figure 2 we fill the latency table of each instruction in order. For example, instruction `i1` has inputs immediately available, so its latency is simply the latency of addition ($L_{\text{add}} = 3$). Instruction `i2` only depends on `i1`'s output (`i2.s1 = i1`), which arrives after `i1` completes plus a cycle of transfer ($A_{\texttt{s1}} = L_{\texttt{i1}} + L_{(\texttt{i1},\texttt{i2})} = 3 + 1 = 4$) and so its final latency including the five cycles of multiplication is $L_{\texttt{mul}} + A_{\texttt{s1}} = 4 + 5 = 9$. Filling the table using Equation 1 reveals that this code snippet takes 15 cycles to complete, with {`i1`, `i4`, `i5`} on the critical path. In our implementation, operation latencies $L_{i.\text{op}}$ are generally stored as constants for immediate operations (`add`, `mul`, etc.) unless time-sharing of functional units is enabled or multiple types of ALU designs are used. Memory access operations are modeled by per-instruction *average memory access time* (AMAT), using counters at load/store unit entries. Data transfer latencies $L_{(i,j)}$ are modeled based on the interconnect used, and measured cycles from counters at individual PEs if available.

3.2 Building the Logical DFG

Figure 3 again shows the main tasks of MESA and the data structures involved: **(T1)** we first build the LDFG from instructions, **(T2)** we spatially map each instruction to form the SDFG, **(T3)** the SDFG is used to configure the accelerator. To build the LDFG, MESA generalizes traditional renaming in out-of-order cores: rather than renaming architectural registers to physical registers, we rename them to instruction addresses. In other words, there are as many physical registers as instructions, which is true in the context of spatial accelerators where each PE produces its own output. Like the case for CPUs, we use a rename table to hold a map of architectural registers to the last instruction that writes to it. In the simple example shown in Figure 3, the first instruction `i1` writes to destination register `r0`, thus `r0` is mapped to `i1` in the rename table. A subsequent instruction with `r0` as source register will thus be replaced with `i1`. This is the case for shown for `i2` in the example. Conceptually, the `r0` data dependency is represented by an edge between `i1` and `i2` in the DFG. The LDFG is built in this manner by simply renaming all instructions in order, and storing collected operation latencies if available. In the first LDFG build, data transfer

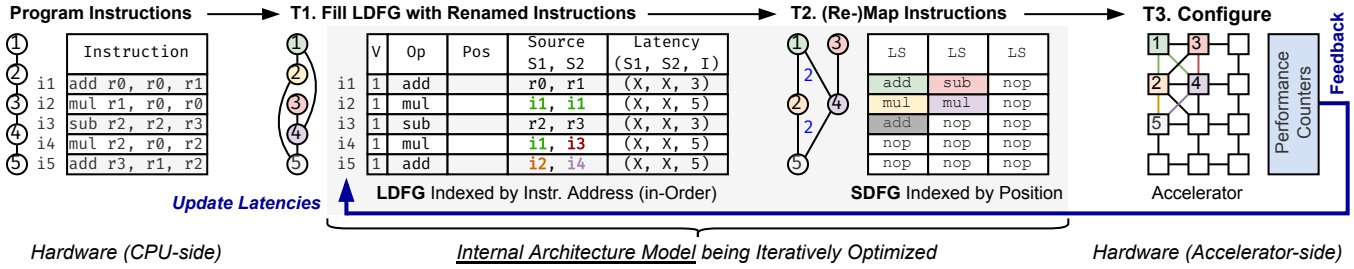


Figure 3: MESA builds and refines a DFG-based architecture model used to optimize and configure the target spatial accelerator.

latencies are not available since we have not mapped operations to the accelerator yet, but this becomes available after performing the mapping algorithm in **(T2)** and in subsequent optimization attempts. The remainder of this section details the proposed spatial mapping algorithm.

3.3 Spatial Mapping Algorithm

The instruction mapping algorithm converts the LDFG to SDFG by assigning each instruction to a coordinate as shown in Figure 3 **(T2)**. The algorithm has a goal of minimizing each instruction's latency as defined in Equation 1. With the DFG performance model available, we have an immediate observation: the latency of an instruction depends solely on the latency of its predecessor with a higher latency (which necessarily lies on the critical path), thus minimizing the transfer latency of this critical path should be of high priority during mapping.

While there is no strict standard, most spatial accelerators today use a dense 2D grid of PEs that are locally connected to their immediate neighbors and globally connected through a shared interconnect to distant units and memories [35]. However, MESA does not restrict the type of interconnect used in the backend as long as it can model the point-to-point communication latency between two PEs. We use a matrix F to represent the placement of instructions in a grid of available functional units (PEs), e.g., assigning F_{ij} = `i2` means instruction `i2` is placed at the PE with virtual coordinate (i, j). The initial state of F is a zero matrix denoting all nops. Note that we use the term 'virtual' since the coordinates here are only used for the spatial DFG model, and will eventually be converted to physical addresses during the configuration step **(T3)**. We also track a free matrix F_{free} which is a binary matrix with the same dimensions as F that keeps track of instruction occupancy, representing the availability of PEs; this is the two-dimensional analog to the register free list for renaming in out-of-order processors. In the case that not all PEs support all operations, a constant masking matrix F_{op} for each operation can be element-wise multiplied (AND) to F_{free} to filter out all occupied or unsupported PEs for the current operation.

Algorithm 1 shows the instruction mapping algorithm used by MESA. For each instruction `i` to be mapped, we consider a candidate matrix C_{i} (a submatrix of F) consisting of nearby positions of its dependents `i.s1` and `i.s2`. For example, for a standard 2D mesh interconnect where latency can be modeled by the Manhattan distance (the number of hops), we can define the candidate submatrix

Algorithm 1: Instruction mapping by minimizing latency.

```
  // Iterate over instructions in LDFG
1 foreach i ∈ {i0, i1, i2,...} do
2     s1 ← i.s1;
3     s2 ← i.s2;
4     C_i ← GenerateCandidateMatrix(i, s1, s2);
      // Filter out unavailable positions
5     C_i ← C_i ⊙ C_free ⊙ C_i.op;
6     minLatency ← ∞;
7     minPosition ← (−1, −1);
      // Determine latency of candidates
8     foreach c ∈ C_i do
9         if c ≠ 0 then
10            A_s1 ← L_s1 + L_(s1,c);
11            A_s2 ← L_s2 + L_(s2,c);
12            expLatency ← L_i.op + max(A_s1, A_s2);
13            if expLatency < minLatency then
14                minLatency ← expLatency;
15                minPosition ← c.pos;
16            end
17        end
18    end
      // Map to latency minimizing position
19    i.pos ← minPosition;
20 end
```

as the equidistant rectangle enclosed by its predecessors as follows.

$$C_{\text{i}} = F_{[\text{s1}_x,\ldots,\text{s2}_x;\,\text{s1}_y,\ldots,\text{s2}_y]} \tag{3}$$

In a hardware implementation, due to the large size of F, we instead determine the candidates from the binary free matrix $C_{\text{i}} = (F_{\text{free}} \odot F_{\text{op}})_{[\text{s1}_x,\ldots,\text{s2}_x;\,\text{s1}_y,\ldots,\text{s2}_y]}$ which is simply a matrix of single bits that can be rapidly accessed. In other words, F_{free} zeroes (filters out) positions that are already occupied, and F_{op} zeroes positions where the PE is incompatible (does not support the current operation). The F_{op} matrices for different operations are predetermined based on the specifications of the hardware backend. In our actual algorithm implementation, however, due to constraints, C_{i} is a fixed 4×8 matrix positioned based on the predecessor with higher latency. We define a latency matrix $l(C_{\text{i}})$ where each element is the instruction's latency L_i if it is placed at that location. Finally, we assign the instruction to the latency minimizing position $\arg\min l(C_{\text{i}})$. If multiple positions have equal latency, we prioritize positions with more free entries in its local neighborhood.

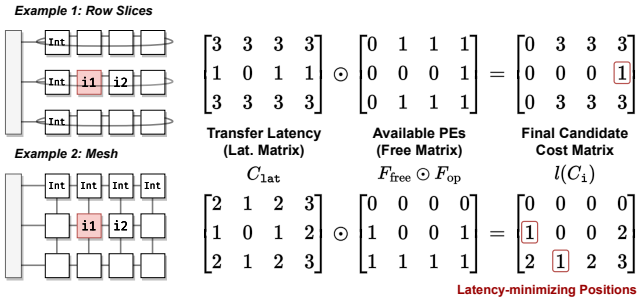


Figure 4: Attempts to place `i3` which solely depends on `i1` as input. In Example 1, the hierarchical interconnect has a 3-cycle data transfer latency across rows but a 1-cycle latency within a row. In Example 2, latency is given by the Manhattan distance. Occupied PEs are filtered out by F_{free} and incompatible PEs are filtered out by F_{op}.

Figure 4 shows an example where we attempt to place `i3` after `i1` and `i2` has already been placed. The same instruction snippet as Figure 3 is used so assume `i3` is an FP multiply that only depends on `i1`. The latency function here is simplified to just be the data transfer latency $L_{(\texttt{i1},\texttt{i3})}$ since $L_{\texttt{i1}}$ is constant and we demonstrate two types of backend interconnects. In Example 1, a hierarchical interconnect of row slices allows point-to-point single-cycle latency between PEs in the same row and a fixed 3-cycle latency across rows. In Example 2, a mesh interconnect has latency equivalent to the Manhattan distance between two points on the grid. The matrices F_{op} filter out integer PEs that are incompatible, and F_{free} filters out positions already occupied by `i1` and `i2`. Finally, the instruction `i3` is placed at a valid (nonzero) position with the smallest cost value. Note that the target cost function $l(C)$ for each position can be more complex than just the estimated data arrival latency, we can add additional metrics to judge how the current mapping affects subsequent unmapped instructions, among other heuristics commonly used by compilers.

Comparison with CGRA Compilers. Compared to typical CGRA compilers [41, 46, 48, 51, 66, 71] MESA follows similar principles but also bears notable differences. For example, MESA performs spatial mapping only and does not time-schedule PEs with multiple instructions. Additionally, MESA performs mapping in a single pass without backtracking. This is why a secondary bus or interconnect should be used as a fallback so that instructions that failed to be mapped can revert back to a slower but less restrictive data forwarding mechanism. In terms of backend architecture support, MESA requires two main components: an operation masking matrix F_{op} provided for each type of operation that indicates which PEs support the operation and a hardware-implementable function $l(C)$ that computes the latency of each position given the current mapping. In other words, the interconnect must be easily modeled such that the latency between any two points can be rapidly calculated. Broadly speaking, MESA follows the same mapping steps as typical CGRA compilers [71] but reduces the complexity of each step due to hardware constraints. For each instruction, MESA gathers a set of candidate PEs available to assign, uses some cost metric (i.e., latency) to enforce an ordering on these candidates, and finally, makes a placement decision based on all available information. While MESA lacks the in-depth considerations and foresight available to compilers, it has a DFG model based on real-time latency data that grants confidence to placement decisions in reflecting actual performance.

4 CONFIGURATION AND OPTIMIZATION

This section describes how MESA detects code regions, handles memory accesses, and configures the accelerator.

4.1 Code Region Detection

There are a number of effective methods for identifying hot program regions [15, 25]. For MESA, we simplify the decision to three conditions that can be tracked during execution. We propose adding a trace-like cache and some non-intrusive instruction monitoring logic to the core frontend to check for the three criteria (**C1-C3**) below that must all be satisfied for MESA to consider the running code for acceleration.

C1: Valid Loop Detection. Loop-stream detection is a technique used in modern high-performance CPUs [56] to detect loops and eliminate redundant fetch and decode of loop instructions. The loop-stream detector (LSD) is typically placed at the decode stage and checks for loops that fit within the CPU's instruction window based on the PC history and explicit jumps or branches with negative offsets. For MESA, the first condition **(C1)** mandates that the loop detected must have fewer instructions than the maximum supported by the accelerator. This is a preliminary check for structural hazards that will arise due to a lack of PEs and load-store entries. The loop's address range (start and end addresses) is recorded by MESA's control registers.

(C2): Control Check. If **(C1)** is met, we enable instruction monitoring at the decode stage to identify unsupported instructions. Unsupported instructions include system instructions (I/O access, `ecalls`, etc.), backward jumps and branches to a target address within the loop (i.e., inner loops), and any instruction type not supported by the target accelerator's functional units (e.g., `64-bit` operations on a `32-bit` accelerator). A tally of instructions checked is tracked at the decode stage, completing this step only when all instructions within the region are checked. A violation invalidates the loop's candidacy for acceleration.

(C3): Instruction Mix. If both **(C1)** and **(C2)** are satisfied, the loop may be accelerable but not necessarily yield promising speedup due to early exit or an unfavorable instruction mix. Thus, concurrent to checking for unsupported instructions, **(C3)** tracks the number of compute and memory instructions relative to loop size. Furthermore, MESA makes an estimate of the loop's expected iteration count based on the branch condition and PC trace. These heuristics are used because acceleration comes at a cost: our evaluation results show that target loops typically need to run 50-100 iterations to offset the initial cost of configuration and offloading, hence it is unwise to proceed without some degree of confidence. Finally, a loop passing all criteria **C1-C3** can still fail to generate an architecture configuration during the mapping process due to failure to route or other structural hazards. In summary, the main microarchitectural additions to the core include an enhanced decode stage with instruction monitoring, a trace cache, MESA control registers, and

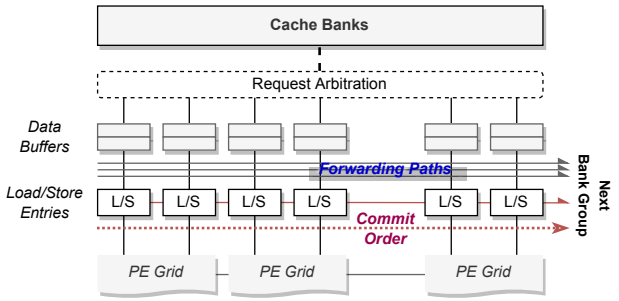


Figure 5: Load/store entries locally interconnected to PEs but maintain original program ordering. Forwarding paths allow stores to broadcast data and address when ready, forwarding data to future loads with matching addresses.

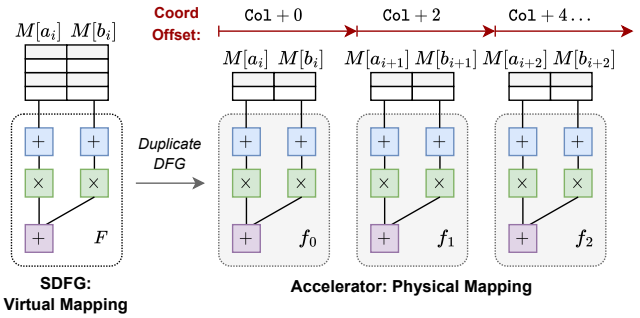


Figure 6: Spatial tiling performed during the configuration step by subgraph duplication, ideally, each instance independently executes in parallel.

some runtime data from branch units and load-store units discussed in the next subsection.

Trace Cache. Like prior work [34, 65], we use an instruction trace cache near the I-cache to store only instructions that are within the code region targeted for acceleration (hence not technically a 'cache'). Instructions fetched from the I-cache are written to the trace cache if their addresses fall within the code region and were not already stored. This trace cache has size equivalent to the maximum number of instructions that can be mapped on the accelerator (condition **(C1)** above), which is 64 - 512 in our evaluations. When MESA builds the LDFG, it accesses the trace cache without interfering with regular fetch on the CPU. In the rare case that, after many profiling iterations, MESA is still missing some instruction(s) in its trace cache, it can temporarily stall the CPU's fetch stage to directly access the I-cache to retrieve missing instructions.

4.2 Memory Access

Unlike the case for compute instructions, MESA's handling of memory accesses depends heavily on the architecture of the spatial accelerator. In its DFG model, memory accesses are abstracted as a node with variable latency. If per-instruction performance counters are available at load-store units, this latency can accurately reflect its average memory access time. Depending on the accelerator's memory subsystem, there are several optimizations we can consider.

Memory Disambiguation. If the accelerator uses traditional load-store queues that enforce ordering (e.g., shared with CPU), memory disambiguation can be performed in much the same way as out-of-order cores. To improve performance, our custom accelerator used for evaluation in later sections is equipped with a more advanced load-store unit that uses the already-built LDFG for ordering and allows data forwarding. Figure 5 shows memory load-store entries connected to PEs (this figure is illustrative only, the actual design has far more entries sharing a port). Recall that the LDFG always maintains the sequence of instructions in original program order, this is important because memory instructions can be assigned and committed (final stores) in original order. However, individual loads can be performed out-of-order as soon as their addresses are generated. Much like LSQs, a load can be invalidated if a prior store instruction commits and matches its address. This invalidation forces the new value to propagate through the remainder of the DFG as if the load had initially been completed.

Store-Load Forwarding. Extraneous store-load pairs to the same addresses can be detected as they have the same address register and offset. Such pairs become a direct forwarding path (an edge in the DFG), thereby eliminating redundant accesses. Forwarding paths shown in Figure 5 allow direct memory data forwarding through data broadcast to identify matching store-load pairs at runtime, eliminating subsequent accesses to the same address.

Vectorization and Prefetching. When MESA builds the LDFG, it tracks changes to the base address register of memory instructions via the rename table as registers are renamed each time they are updated. Load accesses sharing the same (unchanged) base address register with different offsets can be vectorized. Additionally, loads whose base address registers depend only on induction registers can be speculatively prefetched an iteration ahead.

4.3 Accelerator Configuration

MESA configures the spatial accelerator by mapping the SDFG to physical PEs and configuring its interconnect in the process. Recall that the SDFG built by Algorithm 1 is already indexed by coordinate with all parent-to-child connections known. MESA's configuration manager iterates through the SDFG and sends operation and interconnect control bits (a configuration bitstream) to the accelerator. We consider this step virtual to physical mapping as operations assigned to virtual coordinates in the SDFG are mapped to physical locations in hardware. Most PEs are locally connected to their immediate neighbors and since MESA does not time-multiplex PEs, accelerator configuration is only done once per code region unless MESA finds a better mapping in subsequent iterations. Finally, a configuration cache is stored on MESA for loops that have already been mapped in case they are re-encountered in the near future.

Loop-level Optimizations. If a loop is known to be parallelizable without inter-iteration dependencies, then we can apply more advanced loop-level optimizations. As MESA does not speculate at the thread level, this scenario only applies to pre-annotated programs with OpenMP. MESA only supports the pragmas `omp parallel` and `omp simd` where iterations are fully parallelizable without critical sections. As shown in Figure 6, we can fully duplicate instances of the same (virtual) SDFG when configuring the spatial accelerator. Tiling the loop in this manner allows independent DFGs to execute

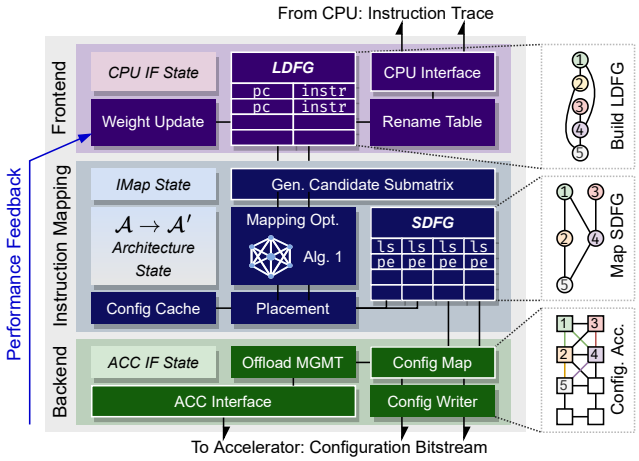


Figure 7: MESA controller's hardware architecture; its frontend interfaces with the CPU and backend interfaces with the accelerator(s).

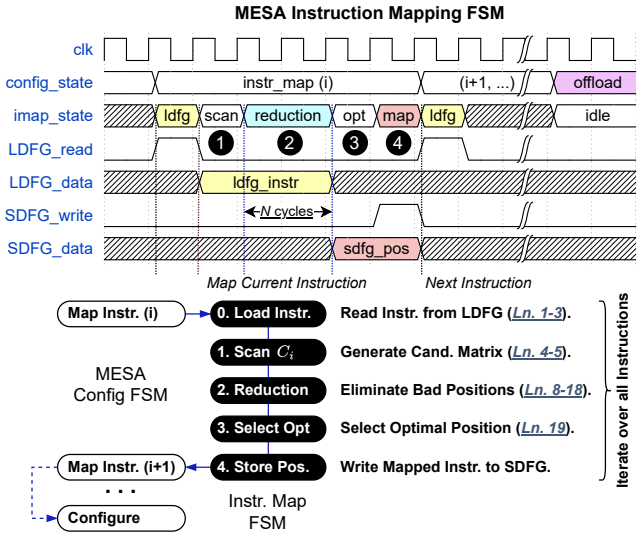


Figure 8: Timing diagram of MESA's instruction mapping state machine. Each stage performs a task corresponding to line(s) of Algorithm 1.

concurrently on the accelerator, greatly improving throughput. Additionally, loop pipelining can also be enabled if supported by the hardware.

5 HARDWARE DESIGN

We implement MESA in synthesizable SystemVerilog, its high-level architecture is shown in Figure 7. MESA's frontend interfaces with the CPU and is mainly responsible for renaming instructions from the trace cache and building the LDFG. Once built, the instruction mapping process can begin: instructions are accessed in order from the LDFG and mapped to the SDFG according to Algorithm 1. Shown in Figure 8 is a timing diagram of instruction mapping stages in the `imap` (InstrMap) state machine. We match the actions of each state with tasks performed in lines of Algorithm 1. In particular, we note that the number of cycles for the reduction stage depends on the dimensions of the candidate matrix, all other states are constant. The `imap` FSM loops until all instructions in the LDFG are mapped to the SDFG. Once the SDFG has been built, MESA's configuration block sequentially writes instructions and routing configuration bits to the accelerator.

5.1 Core Interface

We interfaced MESA with the RISC-V BOOM core [70] under the Chipyard [2] framework with a custom interface to test for control transfer and offloading, it remains future work to have a full MESA-integrated SoC booting an OS. In general, MESA does not negatively interfere with the CPU's regular execution. When a valid code region is detected, the CPU continues executing normally as MESA collects instructions and data from performance counters, if available, to construct the LDFG. When the spatial accelerator is configured, the CPU is allowed to complete its current iteration but is halted when PC reaches the entry point of the accelerated loop or function again; at this point, we wait for all in-flight instructions in the pipeline to commit and transfer control to the accelerator along with the current architectural state (register file, status registers, etc.). During acceleration, the CPU awaits a `return` signal from MESA, and it can context switch in the meantime. When acceleration completes (PC reaches outside the loop region), control is transferred back to the CPU along with the architectural state and a return instruction address from which the CPU resumes much like a subroutine return.

5.2 Accelerator Organization

Though dataflow and CGRA simulators exist, we develop a custom parameterizable spatial accelerator specifically to test various aspects of MESA and to enable end-to-end hardware evaluation. We mainly experiment with three backend configurations: MESA with 128 PEs (M-128) arranged with grid dimension 16×8, of which half are equipped with single-precision floating-point logic; MESA with 512 PEs (M-512), arranged in a 64×8 grid and 64 PEs (M-64) with a 16×4 grid. These dimensions are chosen after evaluating mapping outcomes on different loops. The spatial accelerator uses a hierarchy of execution grids composed of locally-connected functional units arranged geometrically in a 2D mesh. As shown in Figure 9 on the next page, the accelerator has two types of interconnects: local PE-to-PE connections (shown in gray), and a lightweight network-on-chip (shown in blue) which is simply a half-ring interconnect with routing logic at every four PEs, which we call a slice. The transfer latency from a functional unit to its immediate neighbors using direct PE-PE connections is a single cycle. Sending via the on-chip network takes longer depending on traffic and distance but allows long-distance transfers. These two interconnects are used because a poor mapping generated by MESA will result in wasted PEs cycles waiting for data from its parents. The dataflow graphs of loop bodies targeted by MESA are strictly acyclic since MESA does not support nested loops. This means that a mapped loop always has data traveling in a feedforward (topological) fashion from source to destination register; hence each horizontal and vertical lane of the NoC can simply operate like a bus, avoiding possible deadlocks.

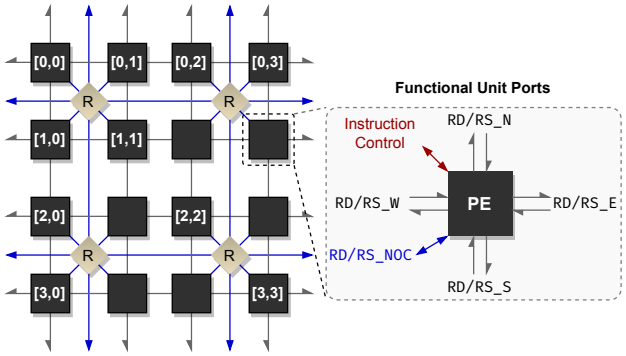


Figure 9: Interconnect used for the test accelerator with direct connections to its neighboring units (gray) and a simple on-chip network (blue) for distant traversals.

Each PE contains an ALU with temporary input/output buffers and registers holding instruction data and address which are assigned during configuration. A comparator between the instruction address and dynamic PC from the control network allows support for forward branches which we discuss below. Finally, a latency counter is placed that tracks the actual L_i, the number of cycles elapsed since the current iteration's start.

Forward Branch Instructions. MESA supports forward branches in the accelerated code region with or without speculation. This is done in a manner similar to predicated execution [39], where instructions (PEs) skipped by a branch can be individually disabled without requiring explicit control transfer. In MESA, instructions under a branch region carry a hidden dependency on the previous instruction producing its destination register, i.e., the instruction previously mapped by the register rename table. This is necessary because, unlike the case for CPUs, disabled PEs must still forward the old register's value as there is no centralized register file. A control unit on the accelerator asserts the enable signal of individual PEs. When a branch is taken, the PEs of all instructions skipped are selectively disabled. On the other hand, backward branches or jumps resulting in inner loops cannot be handled by MESA and must therefore be unrolled by the compiler ahead of time or the loop is disqualified.

Performance Counters. Simple latency counters are placed at PEs and load-store entries on the accelerator to count the start and end cycles of an operation. Note that MESA's DFG model stores node and edge weights, hence these counters track per-instruction latency rather than an averaged IPC or AMAT estimate. These results are reported back to MESA's frontend where latencies are tallied and used to refine MESA's DFG model and used as inputs for future optimization iterations.

5.3 Synthesis Results

We synthesize our hardware design using Synopsys Design Compiler (`R-2020.09-SP4`) with a FreePDK 15nm open cell library [40]. We then perform preliminary place and route of the synthesized design using Synopsys IC Compiler (`L-2016.03-SP1`) for more accurate area and power estimations. Timing for MESA extensions to the CPU is met at 2.0GHz, however, MESA does not necessarily

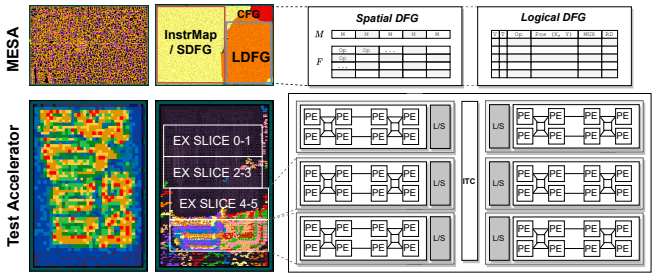


Figure 10: Hardware synthesis of MESA and a custom spatial accelerator (Figure not to scale for visibility, actual area reported in Table 1). MESA's hardware area is primarily dominated by hardware data structures that hold the LDFG and SDFG.

share the CPU core clock domain. The top third of Table 1 shows a hardware area and power consumption breakdown of main components in MESA. Note that we do not have custom physical implementations for large structures like the LDFG and SDFG which are unfortunately synthesized to arrays of registers. We observe that area and power are primarily by the hardware data structures that store the DFG. The middle part of Table 1 shows the total area and power of additional microarchitectural structures to be added to CPU cores for monitoring. The bottom part of Table 1 shows estimates for the custom spatial accelerator used for evaluation. Overall, the overhead of integrating MESA is relatively small

Table 1: Hardware area and power breakdown by component. Synthesis results from Synopsys DC. This table shows a configuration with 128 PEs. (†) Synthesized to register arrays due to lacking SRAM cells. (*) Includes estimations from CACTI[33].

MESA Extensions	Area	Power
MESA Top	0.502 mm^2	0.36 W
|- MESA ArchModel	0.375 mm^2	0.27 W
|-|- Instr. RenameTable	11417.5 μm^2	6.161 mW
|-|- LDFG†	148483.6 μm^2	0.09 W
|-|- Instr. Convert	601.4 μm^2	0.465 mW
|-|- Instr. Mapping	208432.9 μm^2	0.13 W
|-|-|- Latency Optimizer	4060.4 μm^2	3.302 mW
|-|-|- SDFG†	201171.0 μm^2	0.12 W
|- MESA ConfigBlock	101357.9 μm^2	0.07 W
CPU Core Additions	**Area**	**Power**
Trace Cache	27124.5 μm^2	15.455 mW
Add'l Control / Interface	3590.1 μm^2	3.219 mW
Spatial Accelerator	**Area**	**Power**
Accelerator Top*	26.56 mm^2	11.65 W
|- PE Array	14.95 mm^2	4.08 W
|-|- FP Slice (2×2)	821889.1 μm^2	213.107 mW
|-|-|- PE (FP/INT)	204965.6 μm^2	52.229 mW
|-|- INT Slice (2×2)	10415.2 μm^2	18.186 mW
|-|-|- PE (INT)	2337.5 μm^2	3.044 mW

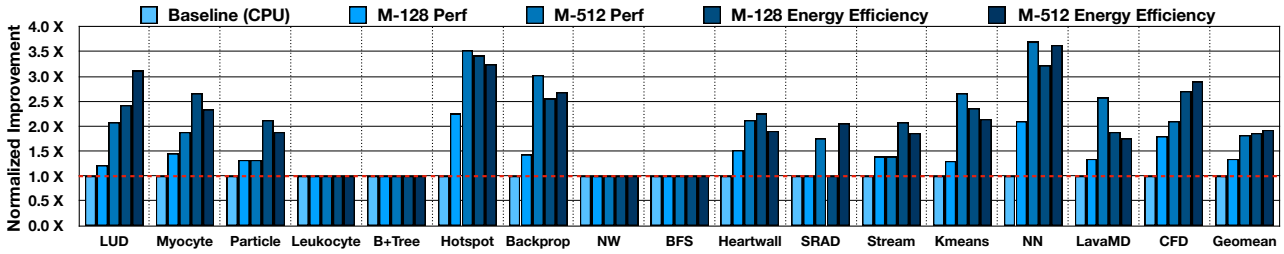


Figure 11: Evaluation results against multicore on the Rodinia benchmark suite in terms of relative performance and energy efficiency.

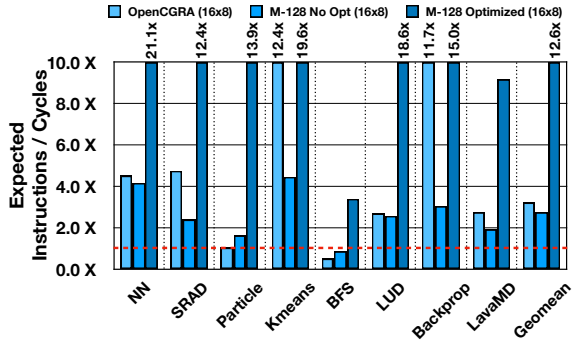


Figure 12: Simulated IPC against a similarly configured OpenCGRA baseline, both without optimizations applied.

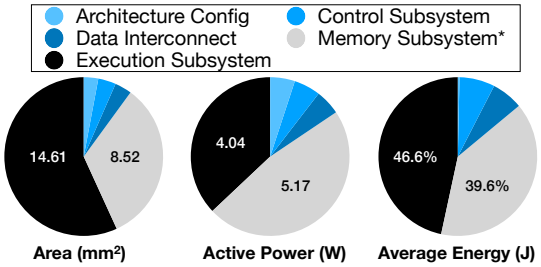


Figure 13: Breakdown of area, power, and energy consumption by component for MESA including the accelerator.

at 0.5mm^2 with negligible per-core additions given that we target accelerator-integrated SoCs. Furthermore, we emphasize that only one MESA controller is needed per chip to interface with all cores unless we explicitly want to configure multiple accelerators simultaneously.

6 EVALUATION

We evaluate MESA's performance and energy efficiency using benchmarks from the Rodinia benchmark suite [8] against a 16-core quad-issue out-of-order RISC-V CPU simulated in *gem5* [5] (based on BOOM as the baseline core). The multicore CPU's area estimates exceed M-128 (26.5mm^2) projecting based on 6mm^2 per core at 28nm [7] to 15nm (we estimate at least >27.5mm^2). Additionally, We also compare MESA against a single-core out-of-order baseline and against a similarly configured CGRA baseline using OpenCGRA [61] as well as DynaSpAM [34].

6.1 Methodology

MESA's performance is evaluated using RTL simulation with QuestaSim 10.4c and Verilator 4.0 for the M-64, M-128, and M-512 configurations. All benchmarks are cross-compiled to target `RV32G` with `-O3` optimizations to run on both the simulated CPU and MESA-enabled accelerator for consistency. For power estimates, we track the activity of PEs in the spatial backend at every cycle in our testbench. A disabled FPU or integer ALU is assumed to be clock-gated and we do not consider its dynamic power. We accumulate the total energy consumed based on the fraction of dynamically active components at every cycle. The *gem5* [5] simulator is used to model the RISC-V CPU baseline that is shared and executes all non-accelerated code. Power is modeled with McPAT [32] by modifying a similarly configured ARM model. A memory hierarchy of 64KB L1, unified 8MB L2 is configured on the simulated system. To achieve manageable simulation speeds, some modules are not simulated with full detail (e.g., FPUs are modeled as constant delays depending on the actual latency of the operation type, which does impact simulation accuracy).

6.2 Performance, Area, and Energy Efficiency

Figure 11 shows the normalized performance and energy efficiency results achieved by M-128 and M-512 against the multicore CPU baseline. On average, MESA achieves 1.33× and 1.81× performance gains across all benchmarks for the two configurations. Note that this average is held back by memory or control-heavy benchmarks like `BFS` that are not suitable for spatial accelerators. In terms of energy efficiency, M-128 and M-512 averaged 1.86× and 1.92× improvement over the CPU respectively. Due to cache limitations, performance does not scale linearly with the number of PEs. In many of these benchmarks, PEs in M-512 are underutilized yielding a result similar to the smaller configuration.

Power Breakdown. A breakdown of total energy consumption is shown in Figure 13, these results are averaged from four benchmarks (`nn, kmeans, hotspot, cfd`). Note that almost 87% of total energy is spent on either memory or computation, with a small fraction on the control subsystem. This is a desirable result as CPU instructions waste significant energy on control overheads [68].

Comparison with OpenCGRA. We compare MESA against a similarly configured CGRA with OpenCGRA [61] in eight Rodinia benchmarks that are compatible. Note that there are fundamental differences between MESA' and OpenCGRA's backends (interconnect latency, temporal scheduling, etc.) which makes a fair comparison difficult, especially since the OpenCGRA tool performs

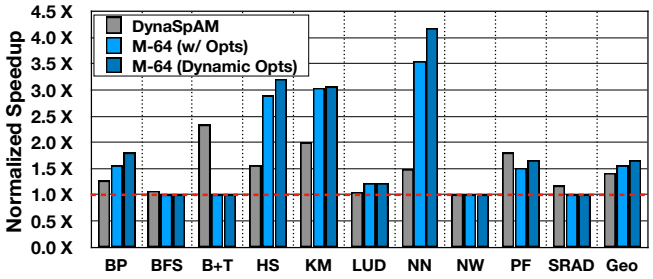


Figure 14: Performance comparison against a single-core baseline and DynaSpAM. We test the smallest M-64 config with optimizations enabled.

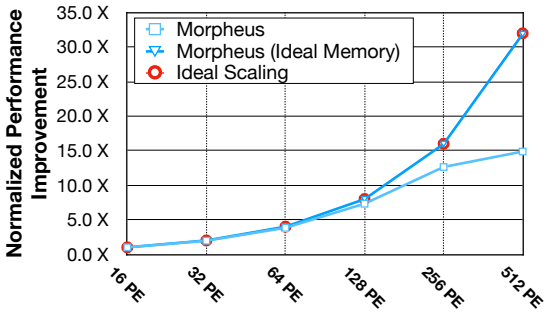


Figure 15: MESA performance scaling with PE count for the `nn` kernel. "Ideal Memory" assumes a scenario with infinite memory ports.

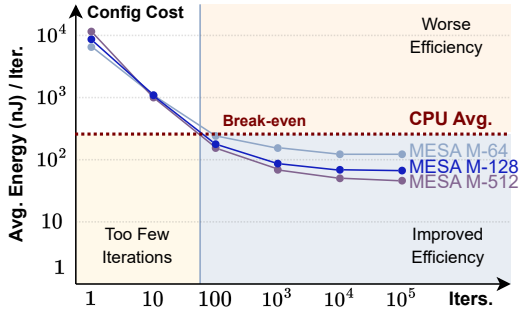


Figure 16: Average energy (nJ) consumed per iteration, the x-axis denotes the number of iterations elapsed.

only CGRA scheduling without optimization features at the time of MESA's development. Thus, we choose to compare two specific aspects of the two methods. First, we disable all optimizations used in MESA to compare only the spatially mapped SDFG against one scheduled by OpenCGRA. We compare the per-iteration IPC: the average number of cycles per loop iteration. Next, we allow MESA to perform its most common optimizations such as tiling, pipelining, etc., and compare the improved IPC. Figure 12 shows that in terms of purely scheduling the operation, MESA falls slightly behind in most benchmarks. This is not a surprise as compiler methods are more complex and expected to generate a better configuration. However, MESA with optimizations enabled easily outperforms OpenCGRA, largely due to enabling loop parallelization.

Comparison with single-core and DynaSpAM. As DynaSpAM is also evaluated on the Rodinia benchmarks, we show a direct comparison with MESA. We repeat our experiments against a single out-of-order core with the *gem5* parameters as listed in the DynaSpAM paper. We use the smallest configuration (M-64) with a synthesized area of 16.4mm^2 for this comparison. Figure 14 shows performance comparisons in their shared benchmarks, DynaSpAM results are with speculation enabled. M-64 with parallel optimizations achieves a speedup of 1.86× compared to DynaSpAM's 1.42×, this increases to 2.01× with runtime iterative reconfiguration. Additionally, since DynaSpAM operates within the core pipeline, there are benchmarks such as `SRAD` and `B+Tree` where the kernel did not qualify for acceleration on MESA. In terms of area efficiency, M-64 falls short compared to a single-core baseline, improving performance by 2.01× but increasing area by an estimated 3.1-4.8×. This is primarily because a small PE array limits the size of accelerable loops and prevents parallel optimizations like tiling, further compounded by MESA's current lack of support for time-multiplexing PEs. The shortage of PE resources is analogous to an out-of-order core having a large instruction window but few functional units, bottlenecking its backend. The next subsection reveals that a PE array either too small or too large leaves MESA of its optimal range.

PE Scaling. We examine in more depth how MESA's mapping effectiveness scales with increasing PE count. Figure 15 shows performance for MESA (default), MESA with an accelerator having unlimited memory ports (ideal memory), and ideal PE scaling. The tested kernel (Euclidean distance) is small enough to fit on just 16 PEs and we observe near-perfect scaling until memory bottlenecks beyond 128 PEs for this spatial accelerator. This also shows that increasing PEs is very cost-effective for exploiting parallelism at least up to M-128. Unlike duplicating CPU cores which carry costly frontend control logic, simply increasing PEs is far more area efficient. This also highlights that too few PEs limits MESA's potential to fully exploit parallelism.

Overheads and Cost Amortization Analysis. The upfront cost of dataflow construction and architecture mapping is also tracked for `nn` as an example. Figure 16 shows the average energy consumed per execution of each loop iteration. Initially, the sunk cost of configuration dominates drastically raises per-iteration energy, however, amortizes over time to around 70 iterations. Acceleration overheads aren't amortized if either too few iterations are run or an inefficient mapping is produced.

Comparison with DBT and Related Approaches. Table 2 shows a comparison of MESA with related approaches. MESA's hardware configuration time is generally between 10^3 and 10^4 cycles, which places it in an interesting middle ground between elaborate compiler-level approaches like DORA and immediate hardware approaches like DynaSpAM. Since we use a hardware implementation, MESA has the capability of targeting 2D spatial architectures despite its short reconfiguration time. This reduction in cost has led to our motivation to allow iterative optimization based on runtime feedback. Results have demonstrated that our method is cost-effective at achieving better speedup while remaining under the microsecond reconfiguration time range.

Table 2: Comparison between MESA and select related research in terms of approach, target hardware, and reconfiguration time.

Work	Config Lat.	Targets	Optimizations
TRIPS [55]	AOT	2D Spatial	H-Block (EDGE)
CCA [12]	-	1D FF	N/A
DynaSpAM [34]	JIT (ns)	1D FF	Out-of-order
DORA [65]	JIT (ms)	2D Spatial	Vect., Unroll, Deepen
MESA	JIT (ns-μs)	2D Spatial	Dynamic, Tile, Pipeline

7 CONCLUSION

To unlock transparent acceleration for general-purpose processors, MESA developed a new method for dynamic binary translation targeting spatial accelerators. MESA extends the CPU's microarchitecture to profile the running application and build a DFG-based model that allows it to model both functionality and performance. We then introduced a data-driven instruction mapping algorithm targeting spatial architectures that is low latency and cost-effective when implemented in hardware. Our implementation shows that MESA requires relatively low area and power investments to add this functionality to the CPU core, and simulation results show promising speedup and efficiency gains. Compared to past works in DBT, MESA finds a balanced middle ground between rapid configuration time and optimization level. A system-on-chip with MESA integrated grants running applications the potential to utilize idle accelerator resources with full transparency. This also allows the accelerator to operate solely in hardware without specialized code or compilers, similar in vein to out-of-order execution in hardware (as opposed to VLIW) but beyond just instruction-level parallelism. Key to MESA's success is that it maintains an internal architecture and performance model of the accelerator, which can be continuously refined. Given the hardware foundation, future work could explore more advanced mapping and optimization strategies with the DFG model and performance data as inputs.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation (CNS-1705047). We would like to thank AMD-Xilinx HACC for their support and for providing research infrastructure. We extend our gratitude towards Cliff Young, James Laudon, and the extended Google Research, Brain Team for their feedback and comments. Nam Sung Kim has a financial interest in Samsung Electronics and NeuroRealityVision.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 105–117. https://doi.org/10.1145/2749469.2750386

[2] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[3] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, and M. Lam. 1987. The Warp Computer: Architecture, Implementation, and Performance. *IEEE Trans. Comput.* 36, 12 (Dec 1987), 1523–1538. https://doi.org/10.1109/TC.1987.5009502

[4] Eunjin Baek, Hunjun Lee, Youngsok Kim, and Jangwoo Kim. 2019. FlexLearn: Fast and Highly Efficient Brain Simulations Using Flexible On-Chip Learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 304–318. https://doi.org/10.1145/3352460.3358268

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (2011), 1–7. https://doi.org/10.1145/2024716.2024718

[6] Arijit Biswas. 2021. Intel Sapphire Rapids. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, New York, NY, USA, 1–22. https://doi.org/10.1109/HCS52781.2021.9566865

[7] Christopher Celio, Pi-Feng Chiu, Krste Asanović, Borivoje Nikolić, and David A. Patterson. 2019. BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS. *IEEE Micro* 39 (2019), 52–60. https://doi.org/10.1109/MM.2019.2897782

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, New York, NY, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[9] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 269–284. https://doi.org/10.1145/2541940.2541967

[10] Tao Chen, Shreesha Srinath, Christopher Batten, and G. Edward Suh. 2018. An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, New York, NY, USA, 55–67. https://doi.org/10.1109/MICRO.2018.00014

[11] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, New York, NY, USA, 367–379. https://doi.org/10.1109/ISCA.2016.40

[12] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. 2004. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture* (Portland, Oregon) *(MICRO 37)*. IEEE Computer Society, USA, 30–40. https://doi.org/10.1109/MICRO.2004.5

[13] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. 2014. A Fully Pipelined and Dynamically Composable Architecture of CGRA. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, New York, NY, USA, 9–16. https://doi.org/10.1109/FCCM.2014.12

[14] 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* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 924–939. https://doi.org/10.1145/3352460.3358276

[15] Evelyn Duesterwald and Vasanth Bala. 2000. Software Profiling for Hot Path Prediction: Less is More. *ASPLOS* 35, 11 (Nov 2000), 202–211. https://doi.org/10.1145/356989.357008

[16] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 365–376. https://doi.org/10.1145/2000064.2000108

[17] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. 2000. PipeRench: A Reconfigurable Architecture and Compiler. *Computer* 33, 4 (apr 2000), 70–77. https://doi.org/10.1109/2.839324

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

[19] Peter Greenhalgh. 2011. Big.LITTLE Processing with ARM Cortex-A15 and Cortex-A7. http://www.arm.com/files/downloads/big.LITTLE_Final.pdf. Accessed: 2022-10-19.

[20] Erika Gunadi and Mikko H. Lipasti. 2011. CRIB: Consolidated Rename, Issue, and Bypass. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/2000064.2000068

[21] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. 2011. Bundled Execution of Recurring Traces for Energy-Efficient General Purpose Processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2155620.2155623

[22] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, New York, NY, USA, Article 56, 13 pages. https://doi.org/10.1109/MICRO.2016.7783759

[23] Mitchell Hayenga, Vignyan Reddy Kothinti Naresh, and Mikko H. Lipasti. 2014. Revolver: Processor architecture for power efficient loop execution. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, New York, NY, USA, 591–602. https://doi.org/10.1109/HPCA.2014.6835968

[24] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

[25] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. 1993. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing* 7, 1–2 (may 1993), 229–248. https://doi.org/10.1007/BF01205185

[26] 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. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[27] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. 2021. A Hardware Accelerator for Protocol Buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 462–478. https://doi.org/10.1145/3466752.3480051

[28] Richard M. Karp and Raymond E. Miller. 1966. Properties of a Model for Parallel Computations: Determinancy, Termination, Queueing. *SIAM J. Appl. Math.* 14, 6 (1966), 1390–1411. https://doi.org/10.1137/0114108

[29] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31, 5 (Sept. 2011), 7–17. https://doi.org/10.1109/MM.2011.89

[30] Ho-Seop Kim and James E. Smith. 2002. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. *SIGARCH Computer Architecture News* 30, 2 (may 2002), 71–81. https://doi.org/10.1145/545214.545224

[31] Chanchal Kumar, Anirudh Seshadri, Aayush Chaudhary, Shubham Bhawalkar, Rohit Singh, and Eric Rotenberg. 2021. Post-Fabrication Microarchitecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1270–1281. https://doi.org/10.1145/3466752.3480119

[32] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[33] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (San Jose, California, USA). IEEE, New York, NY, USA, 694–701. https://doi.org/10.1109/ICCAD.2011.6105405

[34] Feng Liu, Heejin Ahn, Stephen R. Beard, Taewook Oh, and David I. August. 2015. DynaSpAM: Dynamic spatial architecture mapping using Out of Order instruction schedules. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)* (Portland, OR, USA). IEEE, New York, NY, USA, 541–553. https://doi.org/10.1145/2749469.2750414

[35] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *Comput. Surveys* 52, 6, Article 118 (Oct 2019), 39 pages. https://doi.org/10.1145/3357375

[36] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. 2022. OverGen: Improving FPGA Usability through Domain-specific Overlay Generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, New York, NY, USA, 35–56. https://doi.org/10.1109/MICRO56248.2022.00018

[37] Andrea Lottarini, João P. Cerqueira, Thomas J. Repetti, Stephen A. Edwards, Kenneth A. Ross, Mingoo Seok, and Martha A. Kim. 2019. Master of None Acceleration: A Comparison of Accelerator Architectures for Analytical Query Processing. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 762–773. https://doi.org/10.1145/3307650.3322220

[38] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. TABLA: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, New York, NY, USA, 14–26. https://doi.org/10.1109/HPCA.2016.7446050

[39] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. 1995. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (S. Margherita Ligure, Italy) *(ISCA '95)*. Association for Computing Machinery, New York, NY, USA, 138–150. https://doi.org/10.1145/223982.225965

[40] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. 2015. Open Cell Library in 15nm FreePDK Technology. In *ISPD* (Monterey, California, USA) *(ISPD '15)*. Association for Computing Machinery, New York, NY, USA, 171–178. https://doi.org/10.1145/2717764.2717783

[41] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE '03)*. IEEE Computer Society, USA, 10296. https://doi.org/10.1109/DATE.2003.1253623

[42] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2778)*, Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa (Eds.). Springer, Heidelberg, Germany, 61–70. https://doi.org/10.1007/978-3-540-45234-8_7

[43] Mirsky and DeHon. 1996. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE, New York, NY, USA, 157–166. https://doi.org/10.1109/FPGA.1996.564808

[44] 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* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/1168857.1168878

[45] Chris Nicol. 2017. A Coarse Grain Reconfigurable Array (CGRA) for Statically Scheduled Data Flow Computing.

[46] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. 2018. Hybrid Optimization/Heuristic Instruction Scheduling for Programmable Accelerator Codesign. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 36, 15 pages. https://doi.org/10.1145/3243176.3243212

[47] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. *SIGARCH Computer Architecture News* 45, 2 (June 2017), 416–429. https://doi.org/10.1145/3140659.3080255

[48] 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* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 495–506. https://doi.org/10.1145/2491956.2462163

[49] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. 1997. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado, USA) *(ISCA '97)*. Association for Computing Machinery, New York, NY, USA, 206–218.

https://doi.org/10.1145/264107.264201

[50] 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* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 142–153. https://doi.org/10.1145/2485922.2485935

[51] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. 2008. Edge-Centric modulo Scheduling for Coarse-Grained Reconfigurable Architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 166–176. https://doi.org/10.1145/1454115.1454140

[52] Hyunchul Park, Yongjun Park, and Scott Mahlke. 2009. Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 370–380. https://doi.org/10.1145/1669112.1669160

[53] 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* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 389–402. https://doi.org/10.1145/3079856.3080256

[54] Efraim Rotem, Yuli Mandelblat, Vadim Basin, Eli Weissmann, Arik Gihon, Rajshree Chabukswar, Russ Fenger, and Monica Gupta. 2021. Alder Lake Architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, New York, NY, USA, 1–23. https://doi.org/10.1109/HCS52781.2021.9567097

[55] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. 2004. TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP. *ACM Trans. Archit. Code Optim.* 1, 1 (mar 2004), 62–93. https://doi.org/10.1145/980152.980156

[56] Andrey Semin. 2007. Intel Next Generation Nehalem Microarchitecture. https://janth.home.xs4all.nl/HPCourse/PDF/Nehalem_Uarch_TUDelft-Andrey_Semin.pdf. Accessed: 2022-11-02.

[57] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, New York, NY, USA, 97–108. https://doi.org/10.1109/ISCA.2014.6853196

[58] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From High-Level Deep Neural Models to FPGAs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1109/MICRO.2016.7783720

[59] Hartej Singh, Ming-Hau Lee, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, and Eliseu M. Chaves Filho. 2000. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Trans. Comput.* 49, 5 (may 2000), 465–481. https://doi.org/10.1109/12.859540

[60] Johny Srouji. 2020. Introducing M1 Pro and M1 Max: the most powerful chips Apple has ever built. https://www.apple.com/newsroom/2021/10/introducing-m1-pro-and-m1-max-the-most-powerful-chips-apple-has-ever-built. Accessed: 2022-10-19.

[61] Cheng Tan, Chenhao Xie, Ang Li, Kevin J. Barker, and Antonino Tumeo. 2020. OpenCGRA: An Open-Source Unified Framework for Modeling, Testing, and Evaluating CGRAs. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, New York, NY, USA, 381–388. https://doi.org/10.1109/ICCD50377.2020.00070

[62] Brian W. Thompto, Dung Q. Nguyen, José E. Moreira, Ramon Bertran, Hans Jacobson, Richard J. Eickemeyer, Rahul M. Rao, Michael Goulet, Marcy Byers, Christopher J. Gonzalez, Karthik Swaminathan, Nagu R. Dhanwada, Silvia M. Müller, Andreas Wagner, Satish Kumar Sadasivam, Robert K. Montoye, William J. Starke, Christian G. Zoellin, Michael S. Floyd, Jeffrey Stuecheli, Nandhini Chandramoorthy, John-David Wellman, Alper Buyuktosunoglu, Matthias Pflanz, Balaram Sinharoy, and Pradip Bose. 2021. Energy Efficiency Boost in the AI-Infused POWER10 Processor. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, New York, NY, USA, 29–42. https://doi.org/10.1109/ISCA52012.2021.00012

[63] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation Cores: Reducing the Energy of Mature Computations. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 205–218. https://doi.org/10.1145/1736020.1736044

[64] Dong Kai Wang and Nam Sung Kim. 2021. DiAG: A Dataflow-Inspired Architecture for General-Purpose Processors. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 93–106. https://doi.org/10.1145/3445814.3446703

[65] Matthew A. Watkins, Tony Nowatzki, and Anthony Carno. 2016. Software transparent dynamic binary translation for coarse-grain reconfigurable architectures. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, New York, NY, USA, 138–150. https://doi.org/10.1109/HPCA.2016.7446060

[66] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, New York, NY, USA, 268–281. https://doi.org/10.1109/ISCA45697.2020.00032

[67] Ramon Wirsch and Christian Hochberger. 2021. Towards Transparent Dynamic Binary Translation from RISC-V to a CGRA. In *Architecture of Computing Systems: 34th International Conference, ARCS 2021, Virtual Event, June 7–8, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 118–132. https://doi.org/10.1007/978-3-030-81682-7_8

[68] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114

[69] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, New York, NY, USA, 1041–1054. https://doi.org/10.1109/ISCA52012.2021.00085

[70] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine.

[71] Zhongyuan Zhao, Weiguang Sheng, Qin Wang, Wenzhi Yin, Pengfei Ye, Jinchao Li, and Zhigang Mao. 2020. Towards Higher Performance and Robust Compilation for CGRA Modulo Scheduling. *IEEE Transactions on Parallel and Distributed Systems* 31, 9 (2020), 2201–2219. https://doi.org/10.1109/TPDS.2020.2989149