Invited Paper

Building an Open CGRA Ecosystem for Agile Innovation

Rohan Juneja
National University of Singapore
rohan@comp.nus.edu.sg

Pranav Dangi
National University of Singapore
dangi@comp.nus.edu.sg

Thilini Kaushalya Bandara
Renesas Electronics
thilini@comp.nus.edu.sg

Zhaoying Li
National University of Singapore
zhaoying@comp.nus.edu.sg

Dhananjaya Wijerathne
Advanced Micro Devices
dmd.wijerathne@amd.com

Li-Shiuan Peh
National University of Singapore
peh@comp.nus.edu.sg

Tulika Mitra
National University of Singapore
tulika@comp.nus.edu.sg

***Abstract*—Modern computing workloads, particularly in AI and edge applications, demand hardware-software co-design to meet aggressive performance and energy targets. Such co-design benefits from open and agile platforms that replace closed, vertically integrated development with modular, community-driven ecosystems. Coarse-Grained Reconfigurable Architectures (CGRAs), with their unique balance of flexibility and efficiency, are particularly well-suited for this paradigm. When built on open-source hardware generators and software toolchains, CGRAs provide a compelling foundation for architectural exploration, cross-layer optimization, and real-world deployment.**

In this paper, we will present an open CGRA ecosystem that we have developed to support agile innovation across the stack. Our contributions include HyCUBE, a CGRA with a reconfigurable single-cycle multi-hop interconnect for efficient data movement; PACE, which embeds a power-efficient HyCUBE within a RISC-V SoC targeting edge computing; and Morpher, a fully open-source, architecture-adaptive CGRA design framework that supports design space exploration, compilation, simulation, and validation. By embracing openness at every layer, we aim to lower barriers to innovation, enable reproducible research, and demonstrate how CGRAs can anchor the next wave of agile hardware development. We will conclude with a call for a unified abstraction layer for CGRAs and spatial accelerators, one that decouples hardware specialization from software development. Such a representation would unlock architectural portability, compiler innovation, and a scalable, open foundation for spatial computing.

***Index Terms*—CGRAs, Spatial computing, Reconfigurability**

I. Introduction

Modern workloads spanning large-scale deep learning inference (from convolutional neural networks in vision to transformer-based language models), real-time sensor processing in autonomous systems, and ultra-low-power analytics at the edge demand a unique combination of high throughput, tight latency bounds, and extreme energy efficiency (performance-per-watt) that general-purpose processors cannot deliver. Domain-specific accelerators (DSAs) have emerged as an efficient solution for such workloads and are now pervasive in modern SoCs [33]; for instance, Shao et al. [35] report that Apple's SoCs have grown from 10 DSAs in the A4 to over 40 in the A12. However, such specialization leads to dark silicon, as many DSAs remain underutilized across diverse workloads. These applications also exhibit varied computational patterns from dense linear algebra to irregular, data-dependent kernels each requiring distinct dataflows and memory access strategies.

Existing ecosystems are siloed: *The prevailing hardware-software stacks remain tightly coupled and closed, limiting the flexibility and adaptability required for modern workloads.* Commercial GPUs (e.g., NVIDIA's CUDA-driven ecosystem with cuDNN and TensorRT), domain-specific ASICs (e.g., Google's TPU with XLA/TensorFlow integration), FPGA-based dataflow (e.g., Maxeler's proprietary MaxCompiler and runtime), and adaptive compute platforms (e.g., AMD's Versal ACAP with Vitis AI DPU IP) all represent monolithic, vendor-controlled ecosystems. Supporting the diversity in applications is challenging for fixed-function accelerators.

While GPUs and ASICs deliver high throughput and low latency for specific workloads, they limit architectural flexibility, offer closed compilers and rigid execution models, and restrict access to micro-architectural details. FPGA-based platforms offer fine-grained reconfiguration, but still face barriers such as long compilation times due to gate-level synthesis/bit-level reconfiguration and reliance on vendor-specific toolchains. Across all these platforms, inflexible integration and closed-source compilation flows hide critical micro-architectural behavior, hinder third-party compiler innovations, and lock developers into a single vendor's framework. Consequently, architectural exploration is stifled, cross-platform portability is hindered, and tool fragmentation presents hurdles to reproducible, community-driven research. This dynamic elevates vendors to first-class citizens and forces engineers to conform to predefined application profiles. It also amplifies the "hardware lottery" [16] effect by limiting novel research to the narrow subset of ideas

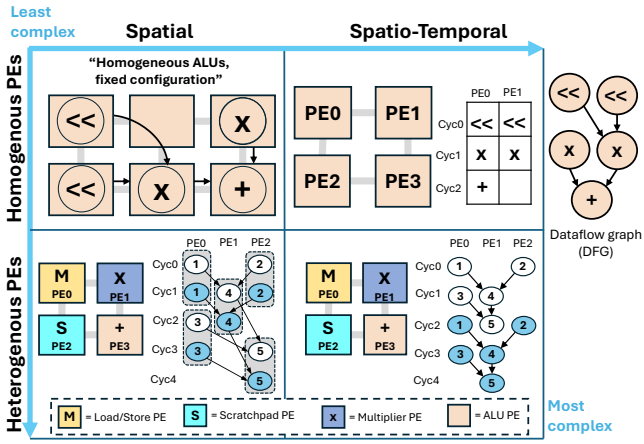

Fig. 1: CGRA Taxonomy.

	Spatial	Spatio-Temporal
Homogenous	MTIA [7], Tensor Core [27], Flex [5]	Renesas DRP [2], Samsung ULP-SRP [19], HyCUBE [18], PACE [25], Nexus Machine [17], Canon [3]
Heterogenous	Warp [1], FPCA [8], Softbrain [26], Tartan [23], Piperench [14], Snafu [12], Riptide [13]	Sambanova RDU [34], Revamp [4], Plaid [20], Amber [11]

TABLE I: Dataflow architectures classified within the CGRA taxonomy. Accelerators often fall under the umbrella of spatial dataflow architectures albeit with minimal reconfiguration.

compatible with existing hardware.

Why Open and Modular platforms? Open and modular ecosystems decouple architectural innovation from monolithic toolchains by offering well-defined interfaces such as an architecture description language (ADL) that allows rapid customization of instruction sets or micro-architectural components without re-engineering the entire compiler stack. These abstractions let hardware architects, compiler developers, and software engineers to work independently, enabling quick and iterative experimentation and validation of new scheduling heuristics, memory hierarchies, or micro-architectural blocks. Configurable platforms shorten this feedback loop by turning infrastructure tweaks or new workload adoption with measurable performance or power gains within hours rather than weeks. This modularity reduces dependence on proprietary toolchains, democratizes hardware-software co-design, and fosters a vibrant, community-driven innovation cycle.

Contributions: In our efforts, we develop and provide an open, modular platform, Morpher [10], that lowers the barrier to CGRA innovation and exploration. Morpher allows researchers to quickly describe new CGRA architectures, compile real applications to them, and validate behavior and evaluate performance. It standardizes the interface between the architecture and compiler, allowing new compositions of processing elements, interconnects, or memory layouts to be explored without rebuilding the toolchain. This enables designs to move from idea to simulation and then to silicon with lowered non-recurring engineering costs. We designed and fabricated open-source HyCUBE CGRA [41] and PACE SoC with CGRA [25]. We release the framework, configurations, and benchmarks to enable fair, reproducible comparisons across dataflow architectures and to enable further research and iteration over our existing work.

II. Background

CGRAs are arrays of programmable processing elements (PEs), each comprising an arithmetic logic unit, a router, and local reconfiguration memory, interconnected by a configurable fabric that maps high-level dataflow graphs at instruction granularity. Data moves between a shared scratchpad memory and the PE array for execution, and results return to the scratchpad on completion. Unlike FPGAs, which configure lookup tables and wires at bit-level to implement arbitrary logic, or domain-specific accelerators, which hard-wire design for specific narrow set of kernels, CGRAs configure functional units and network paths to flexibly support a broad range of workloads with near-ASIC efficiency.

There has been extensive exploration of CGRA architectures from both commercial and academic groups. To organize the design space, we introduce a taxonomy as shown in Fig. 1. The vertical axis contrasts homogeneous arrays of identical PEs with heterogeneous fabrics of specialized units, and the horizontal axis spans spatial mappings (where each operation occupies a PE for its full execution) to spatio-temporal schedules that time multiplex operations across PEs. In the spatial case, the entire DFG can fit onto the CGRA resources without time multiplexing. If the DFG does not fit, the loop body is split into subgraphs, completing all iterations of one subgraph before moving to the next. In the spatio-temporal case, the DFG is mapped across space and time, reconfiguring PEs and interconnects every cycle. Homogeneous arrays are simpler to place and route, while heterogeneous arrays can be more area and energy efficient for matched kernels but are harder to schedule. Overall complexity increases from top left to bottom right.

In Table I, each quadrant includes representative CGRA designs. Moreover, whereas most CGRAs rely on static routing determined at compile time, architectures such as MTIA [7], Nexus Machine [17], Canon [3] support dynamic routing for reconfiguring data paths at runtime, accommodating irregular communication patterns for sparse and graph workloads, enhancing adaptability.

Several open-source CGRA frameworks have been proposed that support modeling, mapping, and evaluation across these features, including CGRA-ME [30], Pillars [15], OpenCGRA [38], and CCF [9]. CGRA-ME provides compiler and RTL for traditional spatio-temporal homogeneous CGRAs but targets simple kernels, lacks control divergence, and only partly handles recurrences; it also omits detailed memory modeling and has no open-source simulator. Pillars adds a Scala description,

Features		CGRA-ME	Pillars	OpenCGRA	CCF	Morpher
DFG Generation	Models control divergence	✗	✗	✓	✓	✓
	Recurrence edges	✗	✗	✓	✓	✓
Architecture Modeling	Adapt user defined architectures	✓	✓	✓	✗	✓
	Multi-hop connections	✗	✗	✗	✗	✓
	Different memory organizations	✗	✗	✓	✗	✓
P&R Mapper	Architecture adaptive mapping	✓	✓	✗	✗	✓
	Data layout aware mapping	✗	✗	✗	✗	✓
	Recurrence aware mapping	✗	✗	✓	✗	✓
Simulation & validation	Cycle accurate simulation	✗	✓	✓	✓	✓
	Test data generation	✗	✗	✗	✗	✓
	Validation against test data	✗	✗	✗	✗	✓

TABLE II: Morpher versus open-source CGRA frameworks.[1]

automatic RTL, and cycle-accurate simulation, but uses CGRA-ME as its frontend and inherit those limitations. OpenCGRA and CCF support control divergence and recurrences, yet their mappers are not architecture adaptive and often need code changes to retarget new interconnects or PE layouts. OpenCGRA's simulator rely on user-written test benches and do not automate test data generation or end-to-end checks. Morpher is open source and automated. It compiles real kernels with control divergence and recurrences, models custom interconnects and memory systems, and uses an architecture-adaptive mapper. It integrates LLVM-based DFG generation and data layout and often yields lower II and faster compile times. It also includes cycle-accurate simulation and automated checking, removing manual benches. Table II compares these frameworks across control, recurrences, adaptivity, interconnects, memory modeling, simulation and validation, and RTL readiness. To our knowledge, it is the only open-source CGRA framework validated end-to-end on fabricated silicon.

III. Open CGRA Infrastrcture

Building on the need for a unified, extensible CGRA toolchain, we have developed Morpher, an open-source, end-to-end CGRA framework that automates architecture generation, mapping, simulation, and validation for both homogeneous and heterogeneous designs. Leveraging Morpher, we have taped out HyCUBE, an 4×4 CGRA featuring a reconfigurable single-cycle, multi-hop mesh that delivers state-of-the-art energy efficiency and throughput, and PACE, a modular RISC-V edge SoC embedding a 8x8 HyCUBE alongside a memory hierarchy and bus interfaces. These silicon prototypes demonstrate Morpher's capability to drive rapid hardware-software co-design. The modular structure of Morpher has also led to the development of several novel CGRA architectures, each reusing certain components from Morpher while adding new components to match their own architectural needs.

A. Morpher

Fig. 2 illustrates Morpher's end-to-end flow, structured into three main phases. In the **Architectural Specification & Data-Flow Extraction** phase, Morpher takes as input ① application source code annotated with the target kernel, ② an ADL description of the CGRA, and ③ a library of Chisel hardware primitives. It then generates the Data-Flow Graph

[1]From Morpher [10].

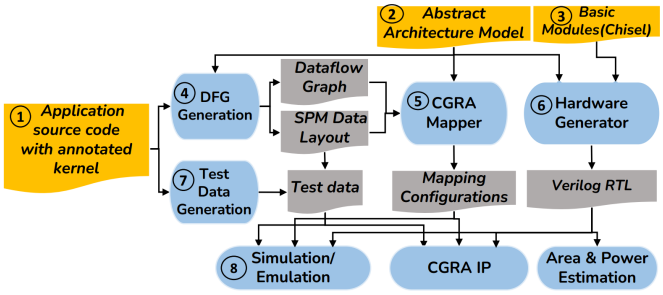

Fig. 2: Morpher framework[1]

and scratchpad memory layout (④) and produces test vectors (⑦) for validation. Next, the **CGRA Mapping** phase maps the extracted DFG onto the CGRA fabric to maximize parallelism by exploiting intra- and inter-iteration parallelism with software pipelining (i.e., modulo scheduling) [31], and output compute and routing configurations. Finally, in the **RTL Generation & Verification** phase, the Hardware Generator (⑥) uses the ADL model to produce Verilog RTL, which the Simulation/Emulation stage (⑧) drives with the test vectors to verify functional correctness and collect area and power estimates.

1) Architectural Specification & Data-Flow Extraction

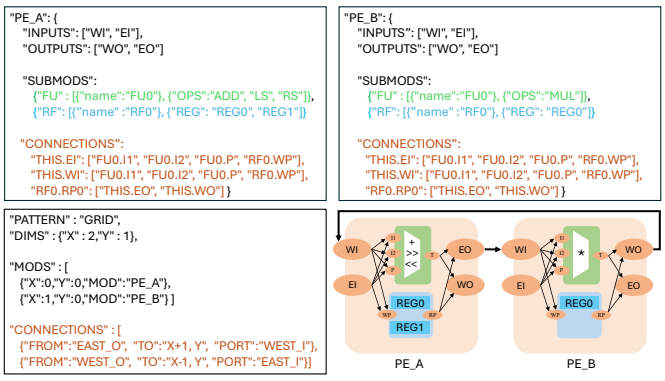

Fig. 3: Example of Morpher ADL for a heterogeneous CGRA with two processing elements. Internal connections of primitive modules (RF, FU) are omitted for simplicity.

Architectural Description Language (ADL): Morpher's ADL provides a concise yet expressive syntax for describing arbitrary CGRA architectures by defining three core abstractions: Modules (hardware blocks such as processing elements (PEs), register files, and memories), Ports (interfaces for connecting producers and consumers), and Connections (interconnect wiring between Ports, including single-cycle, multi-hop links). Generally, all CGRAs can be described by hierarchically structuring the three Morpher ADL's primitive modules: Functional Units (FU), Register Files (RFs), and Memory Units (MU). Multiplexers are are automatically inferred based on the connections between ports. Fig. 3 illustrates this by showing two heterogeneous PE instances, each composed of an FU supporting different operations, an RF, and dedicated input and output ports, linked together.

DFG and Data Layout Generation: Morpher's compiler frontend takes annotated C code as input and generates a

Dataflow Graph (DFG) as output. The C code can come directly from the application or from DSLs like Triton [40] or TVM [6] that can lower to C. Each node in the DFG represents a compute, memory, or predication operation, and includes all relevant metadata, ranging from the opcode and ASAP/ALAP hints for scheduling, to the number of parent nodes and whether the application exhibits any recursive behavior. Each node encodes information about its parent and child nodes as metadata. Depending on the memory access model supported by the target architecture, the DFG generation can be tailored to produce either on-array address computations or decoupled access-execute style addressing, as in the stream-dataflow model [26] with explicitly orchestrated scratchpad banks [28].

Once the DFG is constructed, Morpher also manages data layout into the CGRA's memory banks. For example, in a multi-bank CGRA, it employs simple heuristics like round-robin allocation to minimize contention and memory bank conflicts. Finally, it emits a layout file that records base addresses for arrays and fixed locations for scalars, embedding this address metadata as constants into the corresponding DFG nodes. This data-rich DFG is then passed to the mapper, which parses it to schedule operations onto the actual CGRA fabric.

2) CGRA Mapping

The CGRA Mapper consumes the DFG and the architecture description to generate mapping configurations that minimize the initiation interval (II). II is defined as the number of cycles between the start of consecutive loop iterations and is constrained by resource availability and data dependencies. Starting from the theoretical Minimum II (MII), computed from resource availability and recurrence dependencies, the mapper attempts scheduling with II=MII and increments II until a feasible mapping is found. It first analyzes connectivity constraints between MUs and FUs, annotating each FU with the set of variables it can access. The DFG nodes are then ordered topologically, with recurrence-cycle nodes prioritized by cycle length. Each node is placed onto a space–time instance of the corresponding FU in the Modulo Routing Resource Graph (MRRG), and routed from its parent nodes using Dijkstra's shortest-path algorithm. Ports may be temporarily oversubscribed to improve II convergence.

Once an initial mapping is obtained, the mapper resolves oversubscriptions through one of three strategies: an adaptive heuristic that increases the cost of overused ports (inspired by SPR), a simulated annealing (SA) approach that perturbs node placements along a cooling schedule, or a learning-induced method (LISA) that leverages labels from a trained graph-neural network. The process iterates until all resource conflicts are eliminated. Morpher's modular design makes it straightforward to integrate new mapping algorithms, and future work will extend hierarchical and heterogeneous mapping techniques to improve scalability and support advanced CGRA topologies.

3) Simulation, RTL Generation & Verification

Morpher's ADL is also integrated with a simulation infrastructure, where the simulator parses the ADL and models the corresponding architecture. The simulator accepts the same bitstream and memory layout of the mapped kernel as a prototype CGRA as the input. This setup enables rapid evaluation of both performance and functional correctness of the mappings produced by the compiler.

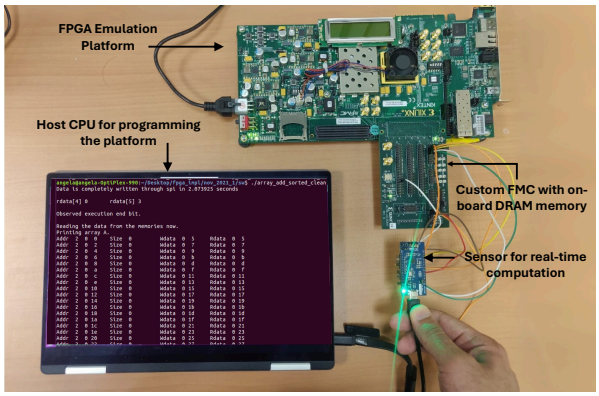

Fig. 4: FPGA emulation of PACE SoC

Finally, Morpher supports RTL generation from the input JSON architecture file. For this, Morpher builds on Pillars [15], utilizing its modular, Chisel-based infrastructure. The JSON file is parsed to instantiate the appropriate Chisel modules, which are then assembled into a complete hardware description and compiled into RTL. This RTL serves as a useful first-pass representation for early-stage area and power estimations, and enables design-space exploration within the Pareto-optimal envelope. However, additional effort is needed to refine this RTL into a concrete and verified implementation suitable for eventual tape-out. Morpher's generated RTL currently supports out-of-the-box FPGA emulation for minor variants of the HyCUBE design. Fig. 4 shows the Morpher-based emulation of a CGRA running a speech detection algorithm on an FPGA with sensors and controlled by a CPU. Nevertheless, automatic generation of valid RTL for arbitrary architectures remains dependent on human intervention, as it is an open challenge in design automation.

B. HyCUBE: CGRA with single-cycle multi-hop interconnects

Traditional CGRAs typically use neighbor-to-neighbor connectivity, where each processing element (PE) can only send data to its immediate north, south, east, or west neighbor. While this keeps the interconnect simple, it severely limits communication flexibility. If an operation needs to send data to a PE more than one hop away, the data must be routed through multiple intermediate PEs, one hop per cycle. These intermediate PEs are congested for routing, even though they're not performing any computation, effectively reducing the available compute parallelism. This type of routing increases the II. Fig. 5(c) illustrates a small loop kernel whose DFG is mapped onto a 2×2 N2N CGRA. Operation *n1* is scheduled on tile *F0* in cycle 0. Its dependents (*n2*, *n3*, *n5*, *n6*) are scheduled over the next few cycles. Due to limited connectivity, the compiler uses F1 and F2 to route the output of *n1* to other PEs. This increases routing pressure, extends the schedule to three cycles per iteration (II = 3), and leaves fewer PEs for actual computation.

[2]Adapted from HyCUBE [18].

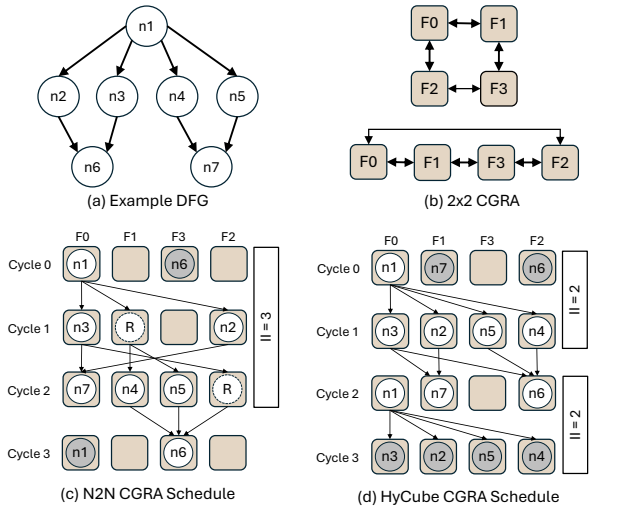

Fig. 5: Mapping of the DFG onto N2N and HyCUBE CGRAs; colored nodes in (c) and (d) indicate operations from the next loop iteration.[2]

In contrast, HyCUBE, shown in Fig. 5(d), introduces a compiler-scheduled, reconfigurable interconnect capable of single-cycle, multi-hop communication. Using clockless repeaters and statically configured crossbars, HyCUBE allows the compiler to set up paths that traverse multiple hops in a single cycle without occupying intermediate PEs. In the same DFG mapped to HyCUBE, *n1* is still on F0 in cycle 0, but now all its dependent operations *n2, n3, n5, and n6* can be scheduled in cycle 1. For example, the edge *n1* to *n5* is routed through the path PEs 0, 1 and 3 in a single cycle. Because the interconnect supports multicast, the same data can be sent to multiple destinations simultaneously without duplication. As a result, the II drops to 2 cycles, improving throughput and PE utilization. HyCUBE was fabricated in a commercial 40nm technology [41]; our test chip delivers a peak energy efficiency of 26.4 MOPS/mW and consumes only 290 pJ per operation. The RTL of HyCUBE is open-sourced to enable further exploration and insight into the architecture.

HyCUBE solves a fundamental bottleneck in CGRA design. **It decouples routing from computation and gives the compiler fine-grained, cycle-level control over communication paths, all while preserving the energy and area efficiency that makes CGRAs attractive in the first place.**

1) Microarchitectural Features

HyCUBE features a 4×4 array of PEs, each equipped with an ALU, local configuration memory, and a crossbar switch. The leftmost column consists of memory-capable tiles that, in addition to computation, include load-store units (LSUs) connected to a shared 4-port data memory, while the remaining tiles are compute-only. The architecture's key innovation lies in its compiler-scheduled interconnect: every PE's crossbar output is driven by clockless repeaters that can be statically configured to bypass or latch data across directions (N, E, W, S). This allows data to travel across multiple PEs in a single cycle without using the PEs in between for routing. The interconnect supports multicast from a single source to multiple destinations within a cycle. This results in an extremely lightweight interconnect, made possible by relying entirely on compiler-determined routes and avoiding the complexity of dynamic routing or flow-control mechanisms. All connectivity is reconfigured cycle-by-cycle via instructions stored in each FU's local configuration memory, making the interconnect highly efficient in both area and power.

HyCUBE eliminates the need for a centralized register file by using distributed registers placed at each directional input. Operands are sent straight to the ALU inputs, or temporarily stored at input registers if needed, which avoids extra move instructions and simplifies control. HyCUBE also supports predication for handling control divergence without requiring explicit predicate registers. Each ALU includes three input registers, two for operands and one for the predicate signal, and each operand includes a predicate flag. Execution occurs only when both predicate in the operands and the predicate input evaluate as valid, and downstream SELECT operations resolve control paths. To support cycle-accurate control, HyCUBE adopts a statically scheduled loop execution model. Thus, each PE's configuration memory is required to store one instruction per cycle over the II, with instruction contents specifying ALU operations, crossbar settings, register accesses, and constants necessary for correct execution.

2) Compiler Support

HyCUBE's reconfigurable interconnect architecture necessitates a compiler that considers cycle-level communication paths in addition to traditional operation scheduling. In modulo scheduling for CGRAs, the compiler constructs a MRRG by unrolling the spatial architecture over a candidate II, replicating FUs, registers, and interconnects across time steps. The MII is computed as the maximum of the resource-constrained (ResMII) and recurrence-constrained (RecMII) bounds [32]. In traditional CGRAs, data dependencies between distant FUs are mapped as multi-cycle paths through intermediate FU nodes, which are temporarily occupied for routing. In contrast, HyCUBE introduces cycle-level reconfigurability in the interconnect, requiring the MRRG to explicitly model links between FUs as schedulable resource nodes. These links are used for single-cycle, multi-hop paths that connect FUs mapping source and sink nodes directly, without involving intermediate FUs.

HyCUBE leverages key components of the Morpher's infrastructure for its compiler-to-hardware integration. The architecture is described using Morpher's ADL primitives. HyCUBE also provides a custom ISA specification, which Morpher uses to automatically generate cycle-accurate configurations and bitstreams for each PE, along with test vectors for functional simulation. Morpher further outputs initial RTL based on parameterized modules, which is used for early evaluation of area and power metrics through commercial synthesis tools. While final deliverable RTL and full backend implementation for tapeout required additional engineering effort, Morpher's modular infrastructure significantly accelerated initial development and system-level validation.

C. PACE: RISC-V SoC with integrated CGRA

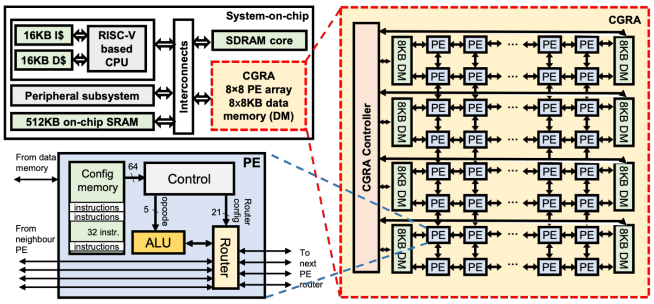

Fig. 6: PACE: CGRA integrated in a RISC-V system-on-chip (SoC)[3]

PACE [24], [25] is a fabricated chip implemented in a commercial 40nm technology node, achieving a peak efficiency of 360 GOPS/W at 0.6V. It extends the HyCUBE architecture by scaling from a single 4×4 CGRA cluster to a 64-PE (8×8) CGRA integrated within a RISC-V-based SoC. The PACE SoC features a 32-bit RISC-V core that manages execution by coordinating data transfers to the CGRA's data and configuration memories, and updates control registers through interrupt-based signaling. Each PE features a 16-bit ALU, 0.25KB of compiler-managed configuration memory, and a statically scheduled, multi-hop crossbar interconnect. Compared to HyCUBE's 32-bit datapaths, all PEs in PACE now use 16-bit datapaths, better aligned with modern AI workload requirements. The CGRA is partitioned into four interconnected clusters, enabling efficient and seamless data communication across cluster boundaries.

To support real-world embedded workloads, the PACE SoC integrates a broad set of external peripherals and memory interfaces. As shown in Fig. 6, the SoC includes an on-chip SRAM and a memory controller that interfaces with an external 32MB SDRAM chip for handling larger data storage requirements. For peripheral interfacing, the SoC includes standard communication protocols such as UART, SPI, and I2C which connect to off-chip components including a bluetooth module, SD card controller, and I2C ROM. Additionally, dedicated ports are provided for general-purpose I/O (GPIO), analog-to-digital conversion (ADC), and cryptographic acceleration via AES and RNG modules. These components are accessed through a shared AXI4 slave multiplexer, providing a uniform MMIO interface to the system. This rich peripheral set enables PACE to support diverse I/O and memory-intensive tasks, making it suitable for real-world edge computing workloads.

PACE further improves its energy efficiency compared to HyCUBE by employing both static and dynamic clock gating. Static clock gating is applied at compile time by identifying periods where specific PEs are idle and disabling their clocks. Complementing this, dynamic clock gating contributes to an additional 10% power reduction through the insertion of idle-state instructions and associated gating logic, as shown in Fig. 7. The compiler inserts NOP instructions to specify intervals during which certain PEs remain idle. These instructions encode the start and end times, allowing local counters within each PE to track the idle period precisely. When the counter is active, the clock to most of the PE's internal logic is gated, excluding critical components like the routing logic that must remain functional. After the counter expires, the PE resumes execution as scheduled. This compiler-coordinated approach allows fine-grained, cycle-accurate clock control that exploits the dataflow execution model, maximizing energy savings.

[3]Adapted from PACE [24], [25].

D. Morpher has enabled broader research in CGRAs

The CGRA design space remains rich with open problems and optimization opportunities across the stack ranging from improved DFG generation tailored to memory access patterns, to enhanced backend compilation involving mapping, placement, and routing, and even the design of novel architectures better suited to specific classes of kernels. Morpher has, in part, enabled several research efforts across this spectrum.

REVAMP [4] is a design-space exploration framework that leverages Morpher's ADL to instantiate heterogeneous CGRA configurations. It employs Morpher's DFG generator to extract dataflow from annotated C code, and extends Morpher's compiler backend with its own customized toolchain. LISA [21] builds on Morpher's compilation flow by replacing its simulated annealing-based mapper with a GNN-based labeling and mapping strategy, generalizable to a range of spatial accelerators and CGRAs. Nexus Machine [17] and Canon [3] both utilize Morpher's DFG generation infrastructure to compile arbitrary application kernels for their respective fabrics. Morpher and HyCUBE have also provided impetus to other CGRA architectures [12], [13], [22], enabling exploration of ideas such as multi-hop routing to improve compilation efficiency and flexibility on their fabrics. CTScan [39] is another interesting work that leverages CGRAs to emulate power side-channels of edge CPUs. It leverages multi-hop capabilities of the PACE CGRA chip to emulate data forwarding across CPU pipeline stages. Prior works in CGRAs [12], [22], [26] have been modeled in Morpher for better insight into various architectural features. Broadly, Morpher's DFG generation, compilation flow, and ADL form a robust research infrastructure, enabling researchers to benchmark, ablate, and contrast their contributions or innovations against prior work.

IV. Experimental Study

This study demonstrates how Morpher can be used to not only accelerate kernels on CGRA-based systems but also is supporting architecture design space exploration. It also highlights the importance of supporting control divergence and recurrence edges.

A. Mapping Quality of Morpher

Benchmark Kernels: We use kernels from a range of popular benchmark suites, including MachSuite, Polybench, Wavelib, and BEEBS, spanning multiple domains and comprising commonly used edge kernels for image processing, filtering, machine learning, and basic linear algebra. Benchmarks with a

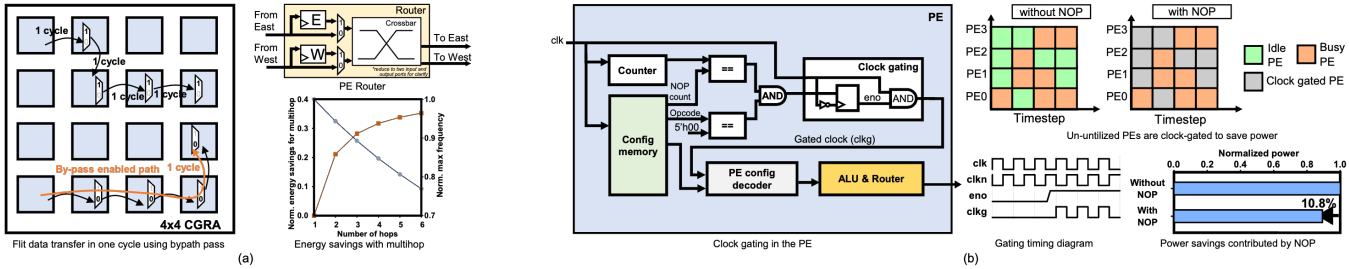

Fig. 7: (a) PE router with bypath pass for single-cycle multi-hop data. (b) Dynamic clock gating for power savings.[3]

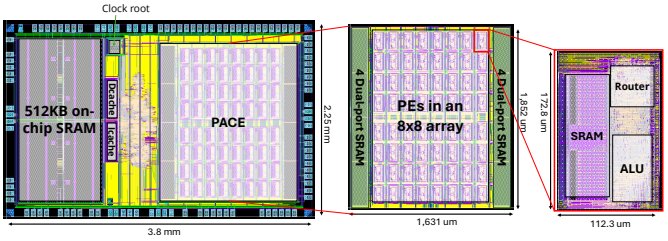

Fig. 8: Modular PACE chip layout, with compact PE layout.[3]

App. Kernels	1-hop	2-hops	3-hops	4-hops
fft	11	5	5	5
adpcm	17	9	9	8
aes	24	15	13	13
disparity	26	12	10	11
dct	23	14	13	13
nw	19	15	15	15
GeMM	14	9	8	7

TABLE III: Impact of multi-hop interconnects on CGRA performance.

small number of graph nodes can be mapped onto the spatial CGRA without partitioning; for such cases, we additionally evaluate unrolled variants (*_u*) to provide better insights.

Baseline Architectures: We evaluate spatial and spatio-temporal CGRA variants from the earlier taxonomy in Fig. 9. The spatial and spatio-temporal architectures are modeled after Snafu [12] and HyCUBE [18], respectively. Across all benchmarks, the spatial architecture exhibits an equal or higher II than the spatio-temporal counterpart, trading some performance for lower power by eliminating configuration memory. Table III quantifies the impact of introducing multi-hop interconnects into CGRA compilation. With two hops, the CGRA already shows performance gains across benchmarks; with four hops, the improvement frequently exceeds 50%, owing to the additional routing freedom and flexibility.

B. PACE SoC evaluations

We measured the PACE silicon across 0.6-1.0 V. CGRA power scales from 4.4 mW at 0.6 V to 43 mW at 1.0 V as shown in Fig. 10(a), while the maximum clock increases from 21 MHz to 105 MHz (Fig. 10(b). Energy efficiency peaks at 360 GOPS/W at 0.6 V/21 MHz and decreases toward ∼154 GOPS/W near 0.95-1.0 V as dynamic power grows faster than throughput (Fig. 10(c). These curves illustrate the expected energy-performance trade-off and motivate operating near 0.6-0.7 V for energy-limited edge deployments, or near 0.9-1.0 V when throughput dominates.

The fabricated SoC (40 nm) occupies 7.6 mm^2, as shown in Fig. 8. As shown in Fig. 11(a), system-level area splits into RISC-V controller 42%, on-chip SRAM 24%, and CGRA 34%. Within the CGRA, area is dominated by PE logic (ALU + control) at 42%, followed by data memory at 29%, PE configuration memory (CM) at 21%, and routing at 8% (Fig. 11(b). Power attribution shows a different balance: CM accounts for 52% of CGRA power, with PE controller 23%, router 14%, ALU 8%, and data memory 3% (see Fig. 11(c). Configuration memory, though modest in area, consumes the most power because the CM is read every cycle to configure ALU, router, and register controls for all PEs.

	Amber [11]	SSCL [37]	ISSCC [36]	JSSC [29]	PACE (Our work)
Year	2022	2020	2019	2020	2023
Tech (nm)	16	28	22	28	40
Area (mm^2)	20.1	3.9	4.9	4.80	3.02
#PEs	384	120	15	64	64
Voltage (V)	1.29	0.6	0.48	0.9	0.6
Freq (MHz)	NA	89	46	800	21
Power (mW)	NA	45.9	NA	537	4.4
Efficiency (GOPS/W)	538	307	978	196	360
Memory	4500KB	234KB	690KB	320KB	80KB
Norm. area (mm^2)[1]	50	5.5	3.2	6.86	3.02
Norm. efficiency (GOPS/W)[2]	86	150	296	96	360

[1] Norm. area = Area $\cdot \frac{\text{node}}{40\text{nm}}$; [2] Norm. efficiency = Efficiency $\cdot \left(\frac{\text{node}}{40\text{nm}}\right)^2$.

TABLE IV: Comparison of this work with prior designs[3].

Table IV compares our CGRA with prior published designs, with efficiency and area normalized to 40 nm to account for process differences. To isolate architectural effects, we disabled workload-dependent power-saving features (e.g., idle-state clock gating) during measurements. Under these conditions, the CGRA achieves 360 GOPS/W at 0.6 V, exceeding prior work by 1.2×-4.6× on the normalized metric, and occupies 3.02 mm^2 (normalized) for the 64-PE array. Beyond peak numbers, the architecture and compiler support a broad mix of kernels and can execute multiple distinct kernels concurrently, indicating greater versatility.

V. Toward a Unified Abstraction Layer for Spatial Accelerators

Morpher seeks to unify a landscape of highly specialized yet fragmented CGRA architectures, but a considerable amount of work remains to fully realize this vision. Its ADL, while detailed, must be extended to capture a wider spectrum of complex and programmable architectures. It remains constrained by a set of supported network topologies, PE types, and memory banking schemes. For instance, the current PE model is limited to a composition of an ALU, a router, and a register file,

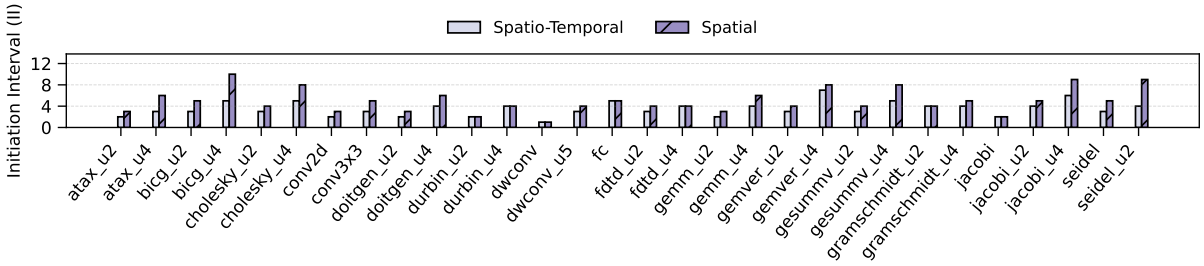

Fig. 9: Summary of benchmark mapping results for spatial and spatio-temporal CGRA architectures.

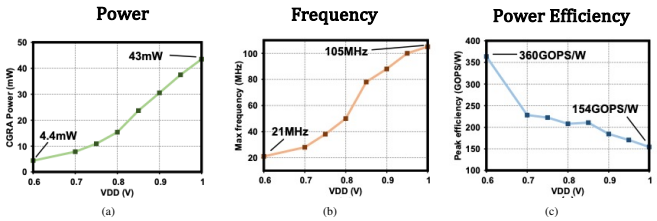

Fig. 10: (a) CGRA power vs. VDD, (b) max frequency vs. VDD, (c) power efficiency vs. VDD.

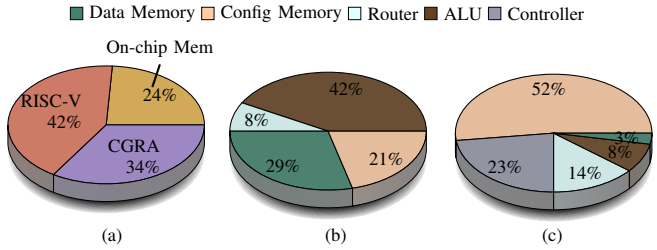

Fig. 11: Area breakdown of (a) the PACE SoC, (b) the CGRA, and (c) power breakdown of CGRA.[3]

whereas more sophisticated architectures may incorporate PEs with various IP blocks, coarser-grained compute cores, caches, or more intricate interconnects. Moreover, the ADL's verbosity imposes a barrier to intuition and creates friction for non-experts attempting to specify architectures.

Morpher framework also lacks programming abstractions that would allow both expert and non-expert users to manually explore architectural or mapping optimizations at a granularity aligned with their expertise. Moreover, its supported architectures primarily target kernels with regular computation patterns and employ a high control-to-compute ratio to preserve generality, resulting in inefficiency relative to domain-specific accelerators.

These challenges motivates a vision for a unified abstraction layer that decouples hardware specialization from software development. In essence, we seek a generalized and common intermediate representation (IR) of spatial computation and communication that remains consistent regardless of the underlying accelerator, allowing software to target a virtual spatial architecture rather than a particular accelerator implementation.

Applications will be first lowered from high-level languages or Domain-Specific Languages (DSL) into this virtual spatial IR. Subsequently the IR is mapped to a particular accelerator through target-specific back ends that perform mapping, scheduling, and routing to generate device configurations/bitstreams. A common IR allows front ends and back ends to evolve independently. The back ends would be parameterized by a declarative architecture specification as an extension of Morpher's ADL of modules, ports, and connections so that the same IR can be retargeted without per-device rewrites.

An architecture independent intermediate representation (IR) for spatial computing would decouple software from device details and enable portability: developers can write kernels once (for example in Triton [40]) and run them on many CGRAs and related accelerators that implement the common IR. Hardware designers can still innovate in PE organization, interconnects, and memory systems without rebuilding the software stack. This separation acts as a stable interface between software and hardware, built on a small set of composable spatial primitives with precise semantics, which extends the lifetime of compilers and tools [43]. A shared IR would also focus compiler work on the common layer, instead of writing new mappers and schedulers for every architecture, as shown by MLIR dialects that decouple CGRA compilation from specific back ends [42]. With this foundation, optimizations for dataflow orchestration, pipelining, and parallel scheduling can be written once and reused across devices.

More importantly, making the abstraction open and modular would establish a scalable foundation for spatial computing. An open standard invites community collaboration: academic and industry contributors could align on interface definitions, contribute to common libraries, and collectively drive improvements. Over time, such a foundation can evolve to incorporate new hardware capabilities (e.g., novel functional units or memory models) while preserving backward compatibility, much as extensible ISAs like RISC-V have done for general-purpose processors. We envision that embracing a unified, open abstraction layer will transform the spatial accelerator landscape into a more portable, innovative, and sustainable ecosystem, combining the efficiency of specialization with the flexibility of a shared, interoperable infrastructure.

VI. Acknowledgments

This research is partially supported by the National Research Foundation, Singapore under its Competitive Research Programme Award NRF-CRP23-2019-0003.

REFERENCES

[1] M. Annaratone *et al.*, "The warp computer: Architecture, implementation, and performance," *IEEE Transactions on Computers*.

[2] "Drp-mpf presentation (final)," http://www.artic.iir.titech.ac.jp/wp/wp-content/uploads/2021/01/DRP-MPF-Presentation_Final.pdf, presentation. Accessed: 2025-08-14.

[3] Z. Bai *et al.*, "A data-driven dynamic execution orchestration architecture," accessed: 2025-08-06. [Online]. Available: https://www.comp.nus.edu.sg/~tulika/ASPLOS26.pdf

[4] T. K. Bandara *et al.*, "Revamp: a systematic framework for heterogeneous cgra realization," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3503222.3507772

[5] T. K. Bandara *et al.*, "Flex: Introducing flexible execution on cgra with spatio-temporal vector dataflow," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023.

[6] T. Chen *et al.*, "Tvm: an automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018.

[7] J. Coburn *et al.*, "Meta's second generation ai chip: Model-chip co-design and productionization experiences," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi-org.libproxy1.nus.edu.sg/10.1145/3695053.3731409

[8] J. Cong *et al.*, "A fully pipelined and dynamically composable architecture of cgra," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014.

[9] S. Dave *et al.*, "Ccf: A cgra compilation framework."

[10] W. Dhananjaya *et al.*, "Morpher: An Open-Source Integrated Compilation and Simulation Framework for CGRA," *Fifth Workshop on Open-Source EDA Technology (WOSET)*.

[11] K. Feng *et al.*, "Amber: A 16-nm system-on-chip with a coarse- grained reconfigurable array for flexible acceleration of dense linear algebra," *IEEE Journal of Solid-State Circuits*.

[12] G. Gobieski *et al.*, "Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture," ser. ISCA '21. IEEE Press, 2021. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00084

[13] G. Gobieski *et al.*, "Riptide: A programmable, energy-minimal dataflow compiler and architecture," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[14] S. Goldstein *et al.*, "Piperench: a coprocessor for streaming multimedia acceleration," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, 1999.

[15] Y. Guo *et al.*, "Pillars: An Integrated CGRA Design Framework," *Third Workshop on Open-Source EDA Technology (WOSET)*.

[16] S. Hooker, "The hardware lottery," *Commun. ACM*. [Online]. Available: https://doi.org/10.1145/3467017

[17] R. Juneja *et al.*, "Nexus machine: An active message inspired reconfigurable architecture for irregular workloads." [Online]. Available: https://arxiv.org/abs/2502.12380

[18] M. Karunaratne *et al.*, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017.

[19] C. Kim *et al.*, "Ulp-srp: Ultra low power samsung reconfigurable processor for biomedical applications," in *2012 International Conference on Field-Programmable Technology*, 2012.

[20] Z. Li *et al.*, "Enhancing cgra efficiency through aligned compute and communication provisioning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.1145/3669940.3707230

[21] Z. Li *et al.*, "Lisa: Graph neural network based portable mapping on spatial accelerators," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[22] J. Melchert *et al.*, "Cascade: An application pipelining toolkit for coarse-grained reconfigurable arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[23] M. Mishra *et al.*, "Tartan: evaluating spatial computation for whole program execution," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, 2006. [Online]. Available: https://doi.org/10.1145/1168857.1168878

[24] V. P. Nambiar *et al.*, "A 360 gops/w cgra in a risc-v soc with multi-hop routers and idle-state instructions for edge computing applications," in *2024 21st International SoC Design Conference (ISOCC)*, 2024.

[25] V. P. Nambiar *et al.*, "Pace: A scalable and energy efficient cgra in a risc-v soc for edge computing applications," in *2024 IEEE Hot Chips 36 Symposium (HCS)*, 2024.

[26] T. Nowatzki *et al.*, "Stream-dataflow acceleration," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[27] NVIDIA Corporation, "Nvidia ampere architecture whitepaper," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, accessed: 2025-07-23.

[28] M. Pellauer *et al.*, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3297858.3304025

[29] G. Peng *et al.*, "A 2.92-gb/s/w and 0.43-gb/s/mg flexible and scalable cgra-based baseband processor for massive mimo detection," *IEEE Journal of Solid-State Circuits*.

[30] O. Ragheb *et al.*, "Cgra-me 2.0: A research framework for next-generation cgra architectures and cad," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2024.

[31] B. R. Rau, "Iterative modulo scheduling," *International Journal of Parallel Programming*.

[32] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: Association for Computing Machinery, 1994. [Online]. Available: https://doi.org/10.1145/192724.192731

[33] V. J. Reddi, "Mobile socs: The wild west of domain-specific architectures," https://www.sigarch.org/mobile-socs/, accessed: 2025-08-13.

[34] "Sn40l rdu ai chip," https://sambanova.ai/products/sn40l-rdu-ai-chip, accessed: 2025-08-14.

[35] Y. S. Shao *et al.*, "RETROSPECTIVE: Aladdin: a pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ISCA@50 25-Year Retrospective: 1996-2020*, J. F. Martínez *et al.*, Eds. ACM SIGARCH and IEEE TCCA, 2023. [Online]. Available: https://bit.ly/isca50_retrospective

[36] S. Smets *et al.*, "2.2 a 978gops/w flexible streaming processor for real-time image processing applications in 22nm fdsoi," in *2019 IEEE International Solid-State Circuits Conference - (ISSCC)*, 2019.

[37] S. Smets *et al.*, "A 28-nm coarse grain 2d-reconfigurable array with data forwarding," *IEEE Solid-State Circuits Letters*.

[38] C. Tan *et al.*, "Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020.

[39] Y. Tavva *et al.*, "Ctscan: A cgra-based platform for the emulation of power side-channel attacks on edge cpus," *ACM Trans. Reconfigurable Technol. Syst.* [Online]. Available: https://doi.org/10.1145/3721294

[40] P. Tillet *et al.*, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3315508.3329973

[41] B. Wang *et al.*, "Hycube: A 0.9v 26.4 mops/mw, 290 pj/op, power efficient accelerator for iot applications," in *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, 2019.

[42] Y. Wang *et al.*, "An mlir-based compilation framework for control flow management on cgras." [Online]. Available: https://arxiv.org/abs/2508.02167

[43] J. Weng *et al.*, "Dsagen: Synthesizing programmable spatial accelerators," in *ISCA 2020*, 2020.