Aquas: Enhancing Domain Specialization through Holistic Hardware-Software Co-Optimization based on MLIR

Yuyang Zou
Peking University
Beijing, China
yyzou25@stu.pku.edu.cn

Youwei Xiao
Peking University
Beijing, China
shallwe@pku.edu.cn

Chenyun Yin
Peking University
Beijing, China
higgs@stu.pku.edu.cn

Yansong Xu
Peking University
Beijing, China
yansongxu.mail@gmail.com

Yuhao Luo
Peking University
Beijing, China
luoyuhao584@gmail.com

Yitian Sun
Peking University
Beijing, China
ytsun@stu.pku.edu.cn

Ruifan Xu
Peking University
Beijing, China
xuruifan@pku.edu.cn

Renze Chen
Peking University
Beijing, China
crz@pku.edu.cn

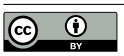
Yun Liang*
School of Integrated Circuits
Peking University
Beijing, China
Beijing Advanced Innovation Center
for Integrated Circuits
Beijing, China
ericlyun@pku.edu.cn

Abstract

Application-Specific Instruction-Set Processors (ASIPs) built on the RISC-V architecture offer specialization opportunities for various applications. Existing frameworks are largely designed around fixed instruction extension interfaces and rely on manual software adaptation. However, as emerging domains scale up in complexity, two major challenges arise. First, memory access remains a primary bottleneck as existing design flows lack architectural awareness of memory interfaces, leading to suboptimal interface selection and orchestration. Second, the semantic complexity of custom instruction extensions, characterized by non-trivial control logic and irregular memory behaviors, hinders the ability of conventional compilers to perform automated and comprehensive offloading.

We present *Aquas*, a holistic hardware-software co-design framework built upon MLIR. *Aquas* proposes a memory interface model that jointly considers interface characteristics and cache effects, with an interface-aware synthesis flow guided by this model that progressively optimizes the input specification and generates efficient hardware implementations. We also propose an e-graph-based retargetable compiler with a novel matching engine that maps and offloads application code to custom instructions robustly and efficiently. Case studies across four diverse domains show that *Aquas* achieves up to 15.61× speedup with 14.5% area overhead and zero frequency degradation, proving highly competitive against more powerful general-purpose cores and vector extensions.

*Corresponding author.



ICCAD '26, San Jose, CA, USA

ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/10.1145/3831252.3834028

CCS Concepts

• **Hardware → Application specific instruction set processors**; *Hardware-software codesign*; • **Software and its engineering → Compilers**.

Keywords

ASIP; Hardware-software co-design; E-graph; Compiler

ACM Reference Format:
Yuyang Zou, Youwei Xiao, Chenyun Yin, Yansong Xu, Yuhao Luo, Yitian Sun, Ruifan Xu, Renze Chen, and Yun Liang. 2026. Aquas: Enhancing Domain Specialization through Holistic Hardware-Software Co-Optimization based on MLIR. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '26), November 08–12, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3831252.3834028

1 Introduction

Edge computing increasingly drives the need for Application-Specific Instruction-Set Processors (ASIPs), which accelerate applications through hardware specialization using instruction set architecture extensions (ISAXs) while maintaining general software programmability. RISC-V [21]-based ASIPs have been widely adopted in signal processing [15], machine learning inference [2, 7], cryptography [4, 12], and computer graphics [18, 19]. Designing high-performance ASIPs presents a significant hardware-software co-design challenge, demanding a framework that can both generate specialized hardware and adapt software to leverage it effectively.

Commercial ASIP tools [17] allow designers to customize processor microarchitecture at the cycle level. Research frameworks [20, 23, 25] identify valuable ISAXs from domain applications, and [11, 13, 26] extend RISC-V cores with instruction extension interfaces like RoCC [1] and CV-X-IF [10], synthesizing execution units via high-level synthesis (HLS). Users then write compiler rules by hand

to offload to these instructions. These excel for scalar or packed SIMD ISAXs, but fall short as emerging domains scale in complexity. Data-intensive workloads such as LLM inference demand not only scaled compute datapaths but also customized memory hierarchies and robust compiler support for complex access patterns, which go beyond the capabilities of existing frameworks.

We identify two indispensable challenges in current ASIP specialization: **Challenge 1: How to fully capture interface characteristics to improve memory access efficiency of ISAXs**. Emerging data-intensive applications impose immense pressure on memory bandwidth. However, data movement in tightly-coupled ASIPs is inherently constrained by the core's memory hierarchy for coherency, and hardware designers often select and orchestrate memory access patterns without systematic consideration of interface capabilities. Commercial tools like Synopsys ASIP Designer [17] offer fine-grained customization, but their cycle-level nature conflates core logic with specific interface timing. This lack of modularity results in a prohibitive engineering overhead. Research frameworks such as Longnail [11] and APS [26] are limited to rigid abstractions of their targeted instruction extension interfaces, lacking flexibility to integrate essential features like local scratchpads, worsening bandwidth, confining them to basic ISAXs.

Challenge 2: How to robustly discover ISAX offloading opportunities to fully exploit the acceleration potential provided by this hardware. The primary obstacle lies in the widening semantic gap between diverse software implementations and increasingly sophisticated ISAX. Application code exhibits significant syntactic divergence despite semantic equivalence, while hardware specialization has evolved to incorporate complex control flow and memory access patterns. This dual-sided complexity leads to an explosion in the program variant space. Current compiler technologies exacerbate this through premature lowering into low-level representations, stripping high-level semantics and introducing implementation-specific noise that exponentially inflates the search space. Consequently, frameworks like APS [26] rely on brittle matching rules that fail under minor code variations, especially loop transformations. Even commercial tools [17] require manual specification between instruction behaviors and C intrinsics.

In this work, we present *Aquas*, a holistic MLIR-based [9] framework for automated hardware-software co-design that generates optimized ASIPs and their corresponding compiler support. On the hardware side, *Aquas* centers on a model of memory-interface attributes and cache effects, with Aquas-IR providing a multi-level intermediate representation (IR) to carry this model through synthesis. The synthesis flow progressively selects access mechanisms and refines transaction granularity before lowering to implementation. For the compiler, *Aquas* overcomes the vulnerability of traditional approaches to syntactic divergence by bridging the hardware-software semantic gap at a structural level. By representing the program within an *e-graph* [22] natively aligned with ISAX semantics, our approach leverages algebraic and control-flow rewrites to non-destructively accumulate a vast equivalence space of program variants. *Aquas* then introduces a hybrid skeleton-components matching engine that strategically navigates this equivalent space to identify mapping candidates robustly and efficiently, unearthing complex offloading opportunities that typically evade conventional compilers. *Aquas* unifies the entire ASIP design flow within the MLIR infrastructure, enabling seamless co-optimization across the hardware-software boundary and facilitating agile design iteration.

Our contributions are as follows:

- We introduce *Aquas*, a holistic open-source framework for unified and automated ASIP hardware-software co-design built upon MLIR infrastructure.
- We propose an interface-aware synthesis flow that models memory interface attributes and automatically generates optimized hardware implementations.
- We design a novel retargetable compiler coupling MLIR's structural representation with *e-graph*-driven equivalence exploration, enabling robust mapping of application code to specialized ISAXs via skeleton-components matching.

We showcase *Aquas* across four case studies: post-quantum cryptography, point-cloud processing, graphics rendering, and CPU LLM inference. It achieves up to 15.61× kernel speedup and 1.95× end-to-end speedup over baseline with less than 22.9% area overhead. Against BOOM [32], a high-performance out-of-order core, and Saturn [31], a RISC-V vector unit, it delivers competitive performance while saving 92.3% and 34% chip area, respectively, demonstrating area efficiency for domain-specific workloads. *Aquas* is open source at https://github.com/pku-liang/aps-mlir.

2 Preliminaries

2.1 Instruction Extension Interfaces

Instruction extension interfaces [1, 5, 10] ease the design of custom instructions without modifying the base processor microarchitecture. They expose the fundamental interfaces for ISAX specialization, including instruction offloading, register-file access, and memory access. Prior frameworks [11, 26] rely entirely on these interfaces and directly generate hardware targeting them.

As emerging workloads such as LLM inference increase memory pressure, interface limitations become a primary bottleneck. These interfaces' memory ports are constrained by limited width and inevitable bubbles, resulting in low effective bandwidth. Designers, therefore, need to route some transfers through higher-bandwidth interfaces such as the system bus, depending on access pattern and performance target. Prior ASIP flows handle this tradeoff with manual, first-glance choices. However, *suboptimal interface selection and deficient orchestration can lead to marked performance loss*, motivating systematic interface-aware memory access optimization.

2.2 MLIR Infrastructure

MLIR [9] provides a powerful solution for optimizing this hardware interface problem. It is a modular compiler infrastructure that allows defining IRs at various abstraction levels and constructing analyses and transformations for optimizations at appropriate levels. Built upon MLIR, CIRCT [6] provides concrete support for handling hardware synthesis tasks. HLS frameworks HECTOR [27], ScaleHLS [29], FESTAL [28], and the ASIP synthesis framework Longnail [11] are built with MLIR.

MLIR's rich expressiveness allows ISAX to encompass complex access patterns and control flows beyond simple dataflow. However, this increased complexity poses a new challenge for software compilers: *Compilers must robustly analyze these complex patterns*

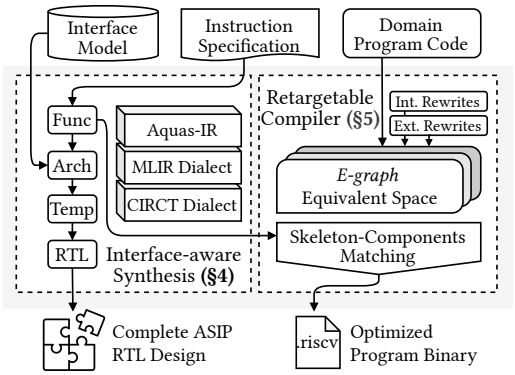


Figure 1: Overview of the unified toolchain in *Aquas*.

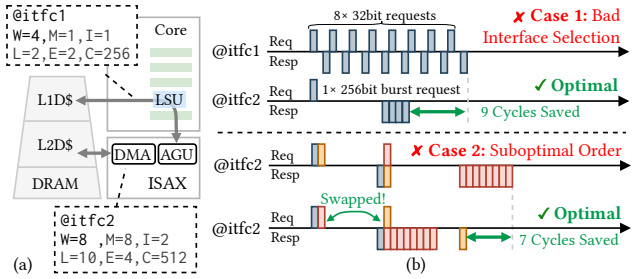


Figure 2: An example of available ISAX memory interfaces in an ASIP and common suboptimal design choices.

to discover as many ISAX offloading opportunities as possible to fully leverage the performance gains. Fortunately, MLIR's layered abstractions allow handling hardware and software at the same level, helping to solve this challenge.

2.3 *E-graph* Representation

E-graphs [30] compactly represent a large number of equivalence relations, serving as a powerful tool to assist MLIR in solving this matching challenge. Within an *e-graph*, *e-classes* group semantically equivalent *e-nodes*. An *e-node* is a function symbol applied to child identifiers, each denoting a child *e-class*. Rewrites match a pattern of *e-nodes* and merge the rewrite results into the original *e-class* through *union* operations. By continuously rewriting, the *e-graph* maintains an equivalence space preserving all rewrite results. An extraction step selects the *e-nodes* minimizing a user-defined cost function to generate a set of optimal expressions.

Overall, we recognize two fundamental challenges in current ASIP specialization: *systematically explore interface-aware synthesis decisions*, and *robustly discover ISAX offloading opportunities*. Our insight is to provide a unified toolchain that models hardware synthesis while solving the software retargeting problem with *e-graph* integration. We leverage various MLIR infrastructures and assets for complete integration.

3 Unified Toolchain

As illustrated in Figure 1, *Aquas* contributes a unified toolchain for agile, flexible, and fully optimized ASIP design. The entire flow integrates MLIR's dialects and assets, introducing new abstractions to build a cross-level synthesis and compilation pipeline. For the hardware-side challenge, we propose a model that jointly considers interface characteristics and cache effects (Section 4.1), and an intermediate representation (*Aquas-IR*) to symbolize and guide decisions during synthesis (Section 4.2). We propose an interface-aware synthesis flow that progressively optimizes the input specification and generates efficient hardware implementation (Section 4.3). For the software-side challenge, we represent the program in MLIR with a similar abstraction level to the ISAX functional behavior (Section 5.1), and use *e-graph* to expand its equivalence space (Section 5.2). By hybrid applying internal and external rewrites under ISAX structure's guidance, we fully expose ISAX offloading opportunities (Section 5.3). Targeting these ISAXs, we apply the skeleton-component matching technique to robustly and accurately locate potential offload points, enabling fully automated retargetable compilation (Section 5.4). This flow enables holistic hardware-software co-optimization for domain specialization.

4 Modeling and Synthesis

This section presents *Aquas*'s interface-aware hardware synthesis, which is structured around a core-ISAX interface model, a progressive multi-level representation (*Aquas-IR*), and a synthesis-time optimization pipeline to generate the final hardware design.

4.1 Modeling Core-ISAX Interfaces

Memory access mechanism selection can have a significant impact on performance. Consider the interfaces depicted in Figure 2(a), `@itfc1` (typically provisioned by an instruction extension interface) offers low latency but is limited to 32-bit width, no burst support, and a single in-flight transaction slot. `@itfc2` provides 64-bit width, burst transfers, and two in-flight transaction slots, but at a higher latency cost. Without analytical guidance, designers are prone to make suboptimal choices. Figure 2(b) shows that improper interface selection or access ordering alone introduces 7–9 cycles of latency penalty, which can severely degrade larger designs.

To systematically analyze these performance implications, we propose a novel core-ISAX interface model. In this model, each memory interface k can be expressed as a 6-tuple:

$$(W_k,\ M_k,\ I_k,\ L_k,\ E_k,\ C_k),$$

where W_k is the interface width in bytes, M_k is the maximum beat count of one transaction, I_k is the maximum in-flight transactions, L_k and E_k denote read lead-off latency and write completion cost, and C_k is the cache-line size visible to that interface.

These interfaces must adhere to specific **microarchitectural constraints**. A transaction of size m is legal if its beat count $m/W_k = 2^t \le M_k$ for some $t \in \mathbb{N}_0$. Additionally, the starting address must be aligned to m, and both read and write transactions support independent pipelining up to the in-flight transaction limit I_k.

Based on this model, the total latency for a sequence of N load or N store transactions on interface k can be estimated as follows. Let m_j be the size in bytes of the j-th transaction. We define a_j as the issue cycle and b_j as the completion cycle of the transaction,

Table 1: Representative operations and exposed microarchitectural features across Aquas-IR abstraction levels.

IR Level	Representative Ops	μ-arch Features
Functional	`transfer, fetch`	m: Specify transfer size
	`read_smem`	
	`read_irf`	
Architectural	`!memitfc<...>`	$W, M_{\max}$: Determine legal length
	`copy # bulk`	I, L, E: Estimate load/store latency
	`load # scalar`	C: Penalize cache refill/flush latency
Temporal	`copy_issue/wait`	I: Determine inflight-aware order
	`load_issue/wait`	Cache hierarchy level: Determine
	`read_smem_issue, read_smem_wait`	phase order and cache locality

with $a_j = b_j = -1$ for $j \le 0$. a_j and b_j are computed as:

$$a_j = 1 + \max(a_{j-1}, b_{j-I_k}),$$
$$b_j^{ld} = \frac{m_j}{W_k} + \max(b_{j-1}, a_j + L_k - 1),$$
$$b_j^{st} = \frac{m_j}{W_k} + E_k + \max(b_{j-1}, a_j - 1).$$

These recurrences serialize transactions waiting for structural slots while overlapping data transfers. The completion cycle b_N gives the estimated interface latency for the sequence.

Cache Hierarchy and Locality. Cache effects are also non-negligible during synthesis, which *Aquas* addresses through two mechanisms. First, cache-line size C_k is incorporated into the model to mitigate cache thrashing. Second, we introduce `cache_hint` annotations within the ISAX specification on memory interfaces and scratchpad buffers to prevent hierarchy mismatches that incur costly synchronization cycles. These hints are readily inferable. For example, in an FIR filter, the large coefficient vector can be tagged `cold` (read from DRAM), while CPU-initialized configuration parameters are tagged `warm` to favor higher-level cache paths.

4.2 Aquas-IR

To integrate this modeling space into the synthesis workflow, we introduce Aquas-IR, organized as three refinement levels, each capturing a specific aspect of synthesis, as summarized in Table 1. At the **functional level**, operations are close to high-level software abstractions. For example, `transfer` and `fetch` are agnostic to the underlying access mechanism, simply specifying the source, destination, and size. The **architectural level** binds memory operations to physical interfaces and legalizes their access patterns. We introduce `!memitfc<>` as module-level symbols encapsulating the physical interface model from Section 4.1. Each `memitfc` symbol carries a unique name and interface parameters W, M, I, L, E and C, which will be used during synthesis to guide interface binding and transfer decomposition. Operations at this level, such as `copy` and `load`, are explicitly bound to one physical interface and annotated with its `memitfc` symbol, making their legality subject to the interface's microarchitectural constraints. Finally, the **temporal level** defines transaction order under in-flight limit I and cache-line size C. Multi-cycle memory latency is modeled via asynchronous issue and wait pairs ordered by each operation's `after` attribute.

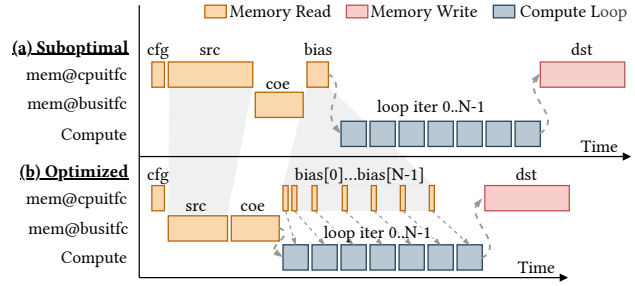


Figure 3: Timing diagram of the `fir7` kernel under a suboptimal lowering and under optimized synthesis pipeline.

```
-memref.global @bias:memref<...>
          {cache_hint = "warm"}
 func.func @fir7(...) {
  scf.for %i=%0 to %48 step %1 {
-   %acc=aquas.read_smem %bias[%i]
+   %acc=aquas.fetch %bi_addr
    // accumulate over 7 taps
    aquas.write_smem %acc,%out[%i]
  }
 } // Functional.mlir                (a)
```

```
+aquas.memitfc @cpuitfc<...>
+aquas.memitfc @busitfc<...>
 func.func @fir7(...) {
- aquas.transfer %cfg_addr,%cfg[0],12
+ aquas.copy @cpuitfc,%c0,%cfg[0],4
+ // 4+4 bytes copies omitted
- aquas.transfer %src_addr,%src[0],108
+ aquas.copy @busitfc,%s0,%src[0],64
+ // 32+8+4 bytes copies omitted
 } {...} // Architectural.mlir        (b)
```

```
func.func @fir7(...) {
- aquas.copy @busitfc, %s0, %src[0], 64
- // Other 32+8+4 bytes copies omitted
+ %t0 = aquas.copy_issue @busitfc, %s96,  %src[96],   8
+ %t1 = aquas.copy_issue @busitfc, %s0,   %src[0],   64 {after = %t0}
+ %t2 = aquas.copy_issue @busitfc, %s64,  %src[64],  32 {after = %t1}
+ %t3 = aquas.copy_issue @busitfc, %s104, %src[104],  4 {after = %t2}
+ aquas.copy_wait %t3
} {...} // Temporal.mlir                (c)
```

Figure 4: A synthesis example for the `fir7` kernel. (a) Scratchpad buffer elision. (b) Interface selection and canonicalization. (c) Transaction scheduling and ordering.

Aquas-IR is implemented as a specialized MLIR dialect, leveraging its robust infrastructure for arithmetic, control flow, and affine transformations to serve as a versatile intermediate target for instruction description languages like CADL[26].

4.3 Synthesis-Time Optimization

Guided by the analytical model, *Aquas* performs interface-aware optimization that progressively optimizes and lowers design specifications through Aquas-IR. Using `fir7` as a running example, where naive manual designs produce suboptimal schedules (shown in Figure 3(a)), we demonstrate how this flow identifies a faster schedule (b), with IR refinements shown in Figure 4.

Scratchpad Buffer Elision. ISAXs often explicitly stage data in local scratchpads. *Aquas* evaluates at the functional level whether intermediate buffers can be safely elided for direct main memory access, reducing latency and SRAM usage. To prevent extra latency or cache degradation, we disable elision for scratchpads accessed within unrolled regions, outside pipelined loops, or used purely as local temporaries. For remaining candidates, affine analysis avoids elisions that trigger cache thrashing. The transformation is accepted only if loop rescheduling confirms no latency increase. In our example (Figure 3), eliding `bias` eliminates bulk transfer latency, and the per-element access of each `bias[i]` is hidden by accumulation

computation, making the elision beneficial. Figure 4(a) shows the resulting IR, where the `bias` declaration is removed and `read_smem` is replaced with a global memory `fetch`.

Interface Selection and Canonicalization. This step lowers functional-level memory operations to the architectural level by solving an optimization problem that assigns each operation to one visible memory interface. Considering all reads or writes separately within one region, for each operation q of m_q bytes, *Aquas* selects one interface k, denoted by the binary variable $X(q,k)$, and greedily splits the request into legal transfer sizes of k in decreasing order. For a properly aligned base operation, this yields an ordered sequence of naturally aligned, legal transfers $\{m_{q,p}^{(k)}\}_p$:

$$\min \sum_k T_k + \sum_{q,k} X(q,k) \left\lceil \frac{m_q}{C_k} \right\rceil \cdot \frac{C_k}{W_k}.$$

where T_k is the estimated transfer latency on interface k based on the model, and the second term penalizes cache hierarchy mismatch by approximating the beat count needed to synchronize the touched cache lines. Here, for each interface k that has operations assigned, T_k can be estimated using an approximation model as follows:

$$T_k = \begin{cases} L_k - 1 + \sum_q X(q,k) \left[\sum_p \max\left(\frac{L_k}{I_k}, \frac{m_{q,p}^{(k)}}{W_k} \right) \right], & \text{if Ops are load} \\ \sum_q X(q,k) \left[\sum_p \left(\frac{m_{q,p}^{(k)}}{W_k} + E_k \right) \right] - 1, & \text{if Ops are store} \end{cases},$$

where $\frac{L_k}{I_k}$ simulates the bubbles introduced due to limited I_k.

In Figure 3, the naive design suboptimally routes the large `src` buffer through the narrow `@cpuitfc`. By evaluating latency and cache penalties, *Aquas* instead assigns `src` to the high-bandwidth `@busitfc`, reducing overall latency. At the IR level (Figure 4(b)), `transfer` operations are lowered to interface-bound `copy` operations. For `src`, the 108-byte transaction is canonicalized into 64-, 32-, 8-, and 4-byte legal transfers to satisfy `@busitfc`'s constraints.

Transaction Scheduling and Ordering. In this step, *Aquas* lowers architectural-level transfers to the temporal level by selecting the order that minimizes completion time under finite I_k and hierarchy constraints. *Aquas* groups transfers by cache hierarchy. Reads closer to the top are issued earlier to prevent cold data from evicting hot data, while writes closer to the bottom are issued earlier to retain hot data in cache longer. Moreover, decomposed segments from the same operation remain contiguous. Within these constraints, *Aquas* finds the minimal-latency schedule via memoized search that compresses exploration state into a relative timing window, exploiting the fact that latency recurrences are insensitive to global time translation. The ordered segments are then lowered into asynchronous primitives, where each transfer becomes a `copy_start` with explicit `after` dependences that capture the chosen issue order.

Figure 4(c) demonstrates the resulting IR transformation. The four decomposed `copy` operations for `src` are optimally reordered and lowered into asynchronous `copy_start` primitives with `after` attributes encoding their execution sequence. A final `copy_wait` synchronizes the last transfer for safe data access.

Hardware Generation. After scheduling, *Aquas* transforms each ISAX into a dynamic pipeline following transactional semantics [8, 24], resolving resource conflicts via arbitration across pipeline stages. *Aquas* then generates backend adapters for the instruction extension interface, along with memory-access engines handling protocol conversion, burst transfers, and misaligned-request fallback. For scratchpad buffers, *Aquas* synthesizes multi-banked memory and address mapping logic. The final design is lowered via CIRCT [6] to FIRRTL and SystemVerilog.

5 Retargetable Compiler

This section presents *Aquas*'s retargetable compilation flow. As shown in Figure 5, it aligns hardware/software to a common abstraction, expands equivalence space via hybrid *e-graph* rewriting, and offloads ISAXs via skeleton-component pattern matching.

5.1 Semantic Alignment

The fundamental obstacle to ISAX matching is the abstraction mismatch between software and hardware representations. Software semantics are expressed in terms of values, control flow, and memory effects, whereas ISAX hardware descriptions contain microarchitecture-specific details such as scratchpad access and register-file interactions. To enable reliable matching, we canonicalize both sides to a common abstraction in base MLIR dialects.

As depicted in Figure 5(❶), software is translated into base MLIR dialects through Polygeist, while hardware instruction descriptions are normalized from the functional level of *Aquas-IR* to the same abstraction. Specifically, **Register operations** like `read_irf` are eliminated by converting explicit register references into direct data dependencies. **Memory operations** like `transfer` are replaced with direct CPU memory accesses, and scratchpad `A` and `C` are removed. The translated ISAX form retains only software-visible control flow and memory effects, comparable to application code.

5.2 Fusing MLIR and *E-graph* Representations

Syntactic divergence makes a single MLIR variant insufficient to cover all matching opportunities. We introduce *e-graph* to compactly represent equivalent variants, with MLIR as the structural backbone and the *e-graph* enabling non-destructive exploration. Our integration maps MLIR structure and transformations into the *e-graph* while preserving *critical semantic relations* such as memory effects and control dependencies. Otherwise, rewrites could violate memory ordering or cross invalid control-flow boundaries, leading to incorrect offloading.

Encoding MLIR Programs into E-graph. We map each MLIR operation to an *e-node* whose children are the *e-classes* of its operands. To preserve execution ordering, we separate operations in each MLIR block into *anchors* and dataflow operations. Anchors are operations that impose strict ordering constraints, including terminators (e.g., `yield`), side-effecting operations (e.g., `store`), and branches. An MLIR block is encoded as a `tuple(...)` e-node with anchors as direct children in program order, and remaining operations form dataflow subtrees attached beneath the anchors that consume their results. Thus, a tuple captures block-level sequencing, while subtrees encode order-independent computations, natively preserving MLIR ordering and dominance within the *e-graph*.

Figure 5(❷) shows the *e-graph* construction. White e-nodes represent the initial *e-graph*. The tuple e-node of each loop region is merged into its corresponding operation, as denoted by bold arrows.

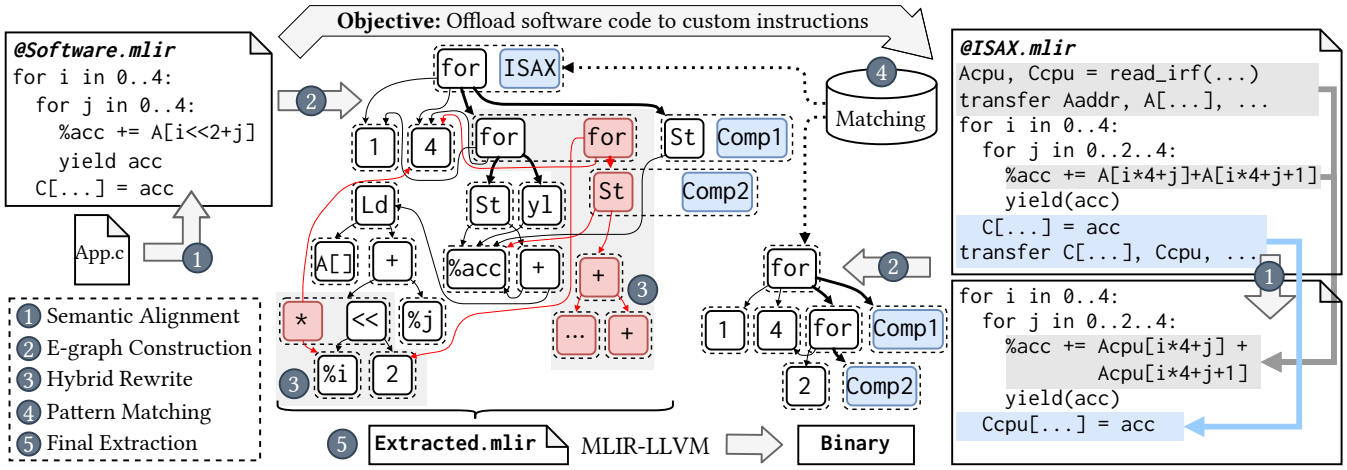


Figure 5: The end-to-end workflow of the *Aquas*'s retargetable compiler.

The child e-nodes of the `for` loop, from left to right, represent the iteration step, the end bound, and the anchors (start bound omitted for brevity). Here, the outermost `for` loop in the software *e-graph* contains two anchors: the nested `for` loop and a `store` e-node.

Reuse MLIR Passes in E-graph. *Aquas* reuses MLIR's pass infrastructure in tandem with the *e-graph*. When an MLIR pass is invoked, the target subgraph is extracted using a cost model to select one optimal e-node per e-class. The extracted subgraph is converted to MLIR operations to compose a valid program, fed into the selected passes, and, upon completion, translated back into the *e-graph* as new e-nodes and unioned with their original e-classes. Overall, the results accumulate in the *e-graph* rather than destructively overwriting previous states, as in a conventional pipeline.

5.3 Hybrid Rewriting for Space Expansion

With *e-graph* representation, we apply rewrites on software code to expand the equivalence space. We propose a hybrid rewriting strategy that iteratively applies two types of rewrites until saturated. **Internal rewrites** apply dataflow transformations such as algebraic simplification on dataflow subtrees beneath anchor e-nodes, leaving control flow structure intact. Since anchors are untouched, control flow and side-effect correctness are preserved. These rewrites are pre-defined as fixed *egglog* [30] rules. **External rewrites** are control-flow-oriented, performing loop transformations such as tiling and unrolling, and are hard to capture as fixed rules, requiring complex checks of dependencies, ordering, and dominance. We implement them via MLIR passes described in Section 5.2.

Blindly saturating external rewrites would cause the *e-graph* to grow explosively due to reshaping program structure, limiting e-node sharing and introducing many new e-classes. To mitigate this, we employ an **ISAX-guided rewrite strategy** that prunes the search space by selectively triggering rewrites based on target loop characteristics, as illustrated in Figure 5(❸). The original software *e-graph* contains a nested loop with a non-affine index (`i≪2`). An internal rewrite already applied `i≪2⇝i*4`. In the external rewrite stage, we analyze the target ISAX's loop characteristics, including iteration counts and stepping conditions. We then enumerate software loop structures and compare their control flow differences with the ISAX. If transformable, we apply the appropriate MLIR passes. The decision here depends only on loop structure without specific operations within the loop body. The cost model is a heuristic scoring function penalizing non-affine operations, steering extraction toward affine-friendly forms (e.g., `i*4` over `i≪2`) for aggressive loop optimization. Here, the inner loop is unrolled by 2 after analysis (differences marked in red). The software *e-graph* now exposes a direct equivalence with the target ISAX.

5.4 Skeleton-Components Pattern Matching

At the final step, we decompose each ISAX into a *skeleton* and several *components* for reliable and scalable matching. The skeleton captures the program's control structure and ordering, enforcing execution order and loop-carried dependencies in each block. Components serve as dataflow subtrees beneath each anchor e-node.

During matching, we first generate *egglog* tagging rules for each component. When a component's subtree matches, a unique marker e-node is inserted into its e-class. Next, a skeleton matching engine identifies candidate e-classes whose enclosing blocks satisfy the required loop/region structure and contain the complete component set, guided by the decomposed instruction skeleton. It then performs a series of checks including ordering, dominance, and dependencies. On success, an `ISAX` marker e-node is inserted. As shown in Figure 5(❹), the ISAX (bottom-right) is decomposed into a two-level loop skeleton and two `components`. The compiler identifies two matching e-classes (tagged with `Comp1`, `Comp2`) in the software *e-graph*. The skeleton engine then selects the outermost `for` node whose loop structure satisfies the skeleton, validates ordering and dependences, and inserts an `ISAX` marker.

Finally, *Aquas* extracts a single optimized program from the *e-graph* (Figure 5(❺)) using a cost model that prioritizes `ISAX` e-nodes. `ISAX` markers are then transformed into intrinsic calls to the corresponding custom instructions. This lowered representation is fed into the standard MLIR-LLVM backend for executable generation, completing the end-to-end compilation flow.

6 Case Studies

We conduct case studies across four domains to evaluate *Aquas* in accelerating real-world workloads and its practical deployability.

6.1 Experiment Setup

We use Rocket core [3] as the base processor, and *Aquas* integrates Chipyard [1] to generate the full ASIP RTL. We enabled two memory access paths, one over the RoCC interface and the other over the system bus. Our baselines include (1) the base Rocket core integrated with ISAXs generated by APS [26], (2) the same core augmented with Saturn [31], and (3) the high-performance out-of-order BOOMv3 core [32] without ISAXs. Area and timing metrics for ASIC are obtained using a commercial tool targeting a 130nm process at the *RocketTile* level, where the baseline Rocket achieves a maximum of 232MHz with an area of 4.11mm^2. We measure performance via RTL simulation with Verilator [16], and report speedup using each core's maximum achievable frequency. All experiments run on a server with 2× AMD EPYC 7763 CPUs and 1.8TiB RAM running Ubuntu 22.04. Both the hardware synthesis flow and the software compilation complete within 2 minutes per case study.

6.2 Post-Quantum Cryptography

Syndrome computation is a core operation in code-based PQC, calculating $\mathbf{s} = \mathbf{H}\mathbf{e}^\top$ over $\mathbb{GF}(2^n)$ to verify or decrypt data using a parity-check matrix $\mathbf{H}$ and error vector $\mathbf{e}$. In practice, $\mathbf{H}$ is typically sparse, and $\mathbf{e}$ from multiple requests can be packed into a matrix to utilize matrix multiplication over $\mathbb{GF}(2^n)$. Accordingly, we design `vdecomp` for bistream unpacking and `mgf2mm` for computation.

Table 2 demonstrates that *Aquas* achieves substantial speedups of 7.59× for `vdecomp` and 3.29× for `mgf2mm` relative to the Rocket baseline. For the end-to-end workload, *Aquas* achieves a 1.42× speedup over Rocket. This performance comes with less than 5.3% area overhead and no frequency degradation.

Memory Access Efficiency. [26] shows limited gains and even a slowdown (0.21×). This disparity is attributed to the fully customized dataflow enabled by *Aquas*'s enhanced memory bandwidth. As [26] lacks block-level memory operations, designers intuitively apply scratchpad buffer elision, leading to severe degradation. *Aquas* avoids this pitfall through interface and access pattern analysis, correctly routing matrix loads to the system bus. These memory access decisions enable *Aquas* to employ more aggressive parallelization strategies, yielding greater speedups.

Compiler Support. The designed ISAXs cannot directly apply to the original software, as they differ in control-flow structure, loop bounds, and data-flow patterns. Transformations such as loop tiling, unrolling, and algebraic rewriting are required to expose matching opportunities. Table 3 lists the detailed differences for each case. The compiler matches all variants, demonstrating the effectiveness of our approach. The matching is fully automated, with *Aquas* iteratively applying internal and external rewrites guided by ISAX loop features. As a result, the *e-graph* size remains manageable, allowing matches within seconds.

6.3 Point-Cloud Processing

The Iterative Closest Point is a fundamental algorithm for 3D registration in point-cloud processing (PCP) [14]. We design ISAXs including *vdist3.vv* (Euclidean distance), *mcov.vs* (covariance matrix), *vfsmax* (maximum comparison), and *vmadot* (matrix-vector multiplication) to accelerate this algorithm. We set the system bus width to 128 bits to evaluate whether *Aquas* can effectively utilize the increased bandwidth via interface-aware mechanisms.

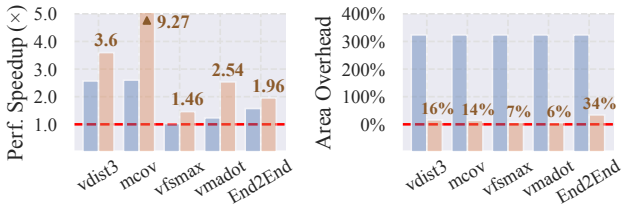


Figure 6: Performance and area comparison between BOOMv3 and *Aquas* on point-cloud processing workloads.

Table 2 details the evaluation results. Compared to the baseline Rocket core, *Aquas* achieves speedups of 1.46×–9.27×, and 1.96× for the full pipeline. The integration incurs no frequency degradation and at most 22.9% area overhead for the end-to-end case, reasonable given the speedup even for edge devices.

Memory Access Efficiency. The algorithm poses significant memory access challenges due to mixed matrix-vector operations and non-2^n length data accesses. APS [26] fails to achieve speedup for the end-to-end case, bottlenecked by *vfsmax* and *vmadot* due to suboptimal memory optimization decisions. *Aquas*, however, effectively utilizes the increased bandwidth through decomposition and scheduling of memory operations, demonstrating the potential for solving real-world design challenges.

Alternative to General-Purpose Cores. Upgrading the core is a common approach to meet performance demands, but a more powerful core such as BOOMv3 incurs 4.24× area overhead and 7.3% frequency drop. We challenge BOOMv3 in domain-specific workloads. Figure 6 shows that *Aquas* can match or exceed BOOM in certain cases with much less area. Compared to BOOM, where memory is bounded by fixed load-store units, *Aquas* leverages customized dataflow to directly exploit extended memory bandwidth and drive extensive hardware parallelism. This proves *Aquas* to be a practical approach with balanced flexibility and performance.

Compiler Robustness. In the end-to-end case, the semantic gap is significantly larger, involving more algebraic transformations that interfere with control flow. As shown in Table 3, the compiler continues to match all patterns. Guided by ISAX loop analysis, loop matching succeeds with limited rewrite passes and keeps the e-node count manageable. In contrast, encoding entire ISAX patterns as monolithic *e-graph* rules failed due to excessive complexity and syntactic brittleness, demonstrating the necessity of our hybrid rewriting strategy, where both internal and external rewrite types are essential and non-interchangeable. The external rewrite mechanism also reuses community-contributed MLIR loop passes, giving greater confidence in correctness.

6.4 Graphics Rendering

Graphics rendering is another important edge computing application. While these workloads are generally accelerated by vector

Table 2: Performance and hardware overheads of custom instructions for PQC and PCP workloads.

Domain	Case	Execution Cycle Counts			Performance Speedup		Minimum Clock Period		Area Overhead	
		Base	ICCAD'25 [26]	***Aquas***	ICCAD'25 [26]	***Aquas***	ICCAD'25 [26]	***Aquas***	ICCAD'25 [26]	***Aquas***
Post-Quantum Cryptography	vdecomp	737	189	97	3.89×	7.59×	+0.2%	+0.0%	+5.6%	+3.2%
	mgf2mm	194	928	59	0.21×	3.29×	+0.0%	+0.0%	+5.3%	+1.6%
	End-to-end	41314	85452	28988	0.48×	1.42×	+0.0%	+0.0%	+11.3%	+5.3%
Point-Cloud Processing	vdist3.vv	495	229	137	2.16×	3.61×	+0.0%	-0.4%	+6.9%	+8.4%
	mcov.vs	853	131	92	6.51×	9.27×	-0.7%	+0.0%	+9.6%	+9.9%
	vfsmax	60	76	41	0.79×	1.46×	+0.0%	+0.0%	+9.6%	+4.9%
	vmadot	127	202	50	0.63×	2.54×	-0.2%	+0.0%	+3.8%	+4.2%
	End-to-end	81466	99372	41596	0.82×	1.96×	-0.2%	+0.0%	+29.6%	+22.9%

Table 3: Compilation statistics.

Case	Control-flow Difference	Dataflow Difference*	Int./Ext. Rewrites	Initial/Saturated e-nodes
vdecomp	Tiling(4)	RF	8/1	298/710
mgf2mm	Unroll(4)	RF, RE	19/1	152/408
End-to-end	Tiling+Unroll +Restructure	RF, RE	20/2	591/1384
vdist3.vv	Tiling(8)	AF, RE	25/1	138/700
mcov.vs	Tiling(4)	AF, RF, RE	24/1	187/556
vfsmax	Tiling+Unroll	RF, RE	15/2	263/1212
vmadot	Unroll(2)	RF, RE	15/1	104/359
End-to-end	Tiling+Unroll	AF, RF, RE	30/3	696/2842

*AF: Algebraic form, RF: Representation form, RE: Common subexpression split/reuse.

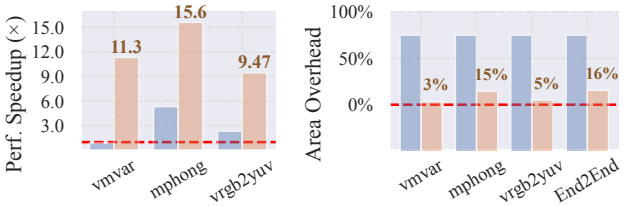
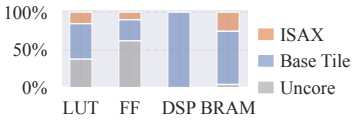


Figure 7: Performance and area comparison between Saturn (RISC-V "V" extension) and *Aquas* on graphics workloads.

instructions like the RISC-V "V" extension, the area induced by such units can be prohibitive for edge applications. We evaluate whether *Aquas* can achieve better performance-area tradeoffs for specific applications. We design three ISAXs, *vmvar* (1st and 2nd vector moments), *mphong* (Phong lighting), and *vrgb2yuv* (color space conversion), compared against Saturn with VLEN=128.

Comparison Result. *Aquas* achieves speedups of 9.47×–15.61× over the baseline, while Saturn achieves 0.91×–5.36×, with a slowdown on *vmvar*. Saturn's poor performance on *vmvar* stems from inefficient reduction operations on such instruction sets. While Saturn's cycle count alone is promising, its integration causes a 35% frequency drop, eroding performance gains and penalizing other parts of the application. *Aquas* also incurs much smaller area overhead (15.6% vs. 75% for Saturn). Even excluding Saturn's unused floating-point unit, *Aquas* still saves over 26% area. Overall, *Aquas* achieves a better performance-overhead balance, making it the preferable choice for domain specialization.



(a) Resource breakdown

Platform	TTFT(s)	ITL(s)
Base	32.00	32.15
Aquas	3.44	3.52

(b) Latency results

Figure 8: FPGA evaluation results on CPU LLM inference.

6.5 CPU LLM Inference

We prototype *Aquas* on an FPGA board with AMD Xilinx XC7Z045 (350K logic cells, 17.6Mb BRAM, 900 DSP slices) to validate real-world performance and hardware efficiency for edge-based LLM inference. We design a set of ISAXs to accelerate attention computation. Both cores run at 80MHz with 1GB DDR3 DRAM, simulating real-world edge scenarios. Our target model is Llama 2 with 110M-parameter and 8-bit quantization. We use Vivado version 2024.1.

Resource Usage and Performance. Figure 8(a) shows the resource breakout of the SoC. The custom instruction accounts for 15% LUT, 10% FF, and 25% BRAM usage, where the higher BRAM utilization is dedicated to scratchpad buffers to mitigate the off-chip memory access bottleneck. This modest resource investment translates into significant performance gains: *Aquas* achieves 9.30× and 9.13× speedup in time-to-first-token (TTFT) and inter-token latency (ITL) over the baseline, respectively, as shown in Figure 8(b). This speedup stems from efficient memory accesses and the highly parallelized datapath they enable. These results demonstrate *Aquas*'s real-world effectiveness and the viability of physical implementation. We believe that with more aggressive quantization and sparsity, *Aquas* can achieve even greater speedups.

7 Conclusion

This paper presents *Aquas*, an ASIP hardware-software co-design framework built upon MLIR. It introduces an interface-aware synthesis flow guided by memory-interface to generate efficient hardware, and an novel *e-graph*-based retargetable compiler for automatic ISAX mapping. *Aquas* achieves up to 15.61× kernel and 1.95× end-to-end speedups with minor frequency and area overheads.

Acknowledgments

This work was supported in part by the National Science Foundation of China (Grant No. T2325001).

References

[1] 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. doi:10.1109/MM.2020.2996616

[2] Giorgos Armeniakos, Alexis Maras, Sotirios Xydis, and Dimitrios Soudris. 2024. Mixed-precision Neural Networks on RISC-V Cores: ISA Extensions for Multi-Pumped Soft SIMD Operations. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. Association for Computing Machinery, Newark, NJ, USA, Article 235, 9 pages. doi:10.1145/3676536.3676840

[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Berkeley, CA. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[4] Hao Cheng, Georgios Fotiadis, Johann Großschädl, Daniel Page, Thinh H. Pham, and Peter Y. A. Ryan. 2024. RISC-V Instruction Set Extensions for Multi-Precision Integer Arithmetic. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC '24)*. Association for Computing Machinery, San Francisco, CA, USA, 1–6. doi:10.1145/3649329.3657347

[5] Mihaela Damian, Julian Oppermann, Christoph Spang, and Andreas Koch. 2022. SCAIE-V: an open-source SCAlable interface for ISA extensions for RISC-V processors. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. ACM, San Francisco, CA, USA, 169–174. doi:10.1145/3489517.3530432

[6] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, Andrew Young, and Richard Xia. 2021. MLIR as hardware compiler infrastructure. In *Workshop on Open-Source EDA Technology (WOSET)*, Vol. 3. Virtual Event, 1 pages. https://woset-workshop.github.io/PDFs/2021/a06.pdf

[7] Gerasimos Gerogiannis, Stijn Eyerman, Evangelos Georganas, Wim Heirman, and Josep Torrellas. 2025. DECA: A Near-Core LLM Decompression Accelerator Grounded on a 3D Roofline Model. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, Athens, Greece, 184–200. doi:10.1145/3725843.3756073

[8] James C. Hoe and Arvind. 2000. Synthesis of operation-centric hardware descriptions. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD - 2000. IEEE/ACM Digest of Technical Papers (Cat. No.00CH37140) (ICCAD '00)*. IEEE Press, San Jose, CA, USA, 511–518. doi:10.1109/ICCAD.2000.896524

[9] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Seoul, Republic of Korea, 2–14. doi:10.1109/CGO51591.2021.9370308

[10] OpenHW Group. 2026. openhwgroup/core-v-xif. original-date: 2021-03-04T12:54:16Z. https://github.com/openhwgroup/core-v-xif

[11] Julian Oppermann, Brindusa Mihaela Damian-Kosterhon, Florian Meisel, Tammo Mürmann, Eyck Jentzsch, and Andreas Koch. 2024. Longnail: High-Level Synthesis of Portable Custom Instruction Set Extensions for RISC-V Processors from Descriptions in the Open-Source CoreDSL Language. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ACM, La Jolla, CA, USA, 591–606. doi:10.1145/3620666.3651375

[12] Tianwei Pan, Tianao Dai, Jianlei Yang, Hongbin Jing, Yang Su, Zeyu Hao, Xiaotao Jia, Chunming Hu, and Weisheng Zhao. 2025. Finesse: An Agile Design Framework for Pairing-based Cryptography via Software/Hardware Co-Design. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. ACM, Tokyo, Japan, 65–77. doi:10.1145/3695053.3731033

[13] Weijie Peng, Youwei Xiao, Yuyang Zou, Zizhang Luo, and Yun Liang. 2025. Clay: High-level ASIP Framework for Flexible Microarchitecture-Aware Instruction Customization. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, Munich, Germany, 1–9. ISSN: 1558-2434. doi:10.1109/ICCAD66269.2025.11240669

[14] Radu Bogdan Rusu and Steve Cousins. 2011. 3D is here: Point Cloud Library (PCL). In *2011 IEEE International Conference on Robotics and Automation*. IEEE, Shanghai, China, 1–4. doi:10.1109/ICRA.2011.5980567

[15] Paul Scheffler, Luca Colagrande, and Luca Benini. 2024. SARIS: Accelerating Stencil Computations on Energy-Efficient RISC-V Compute Clusters with Indirect Stream Registers. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC '24)*. Association for Computing Machinery, San Francisco, CA, USA, 1–6. doi:10.1145/3649329.3658494

[16] Wilson Snyder, Paul Wasson, Duane Galbi, and Geza Lore. 2025. *Verilator*. https://verilator.org

[17] Synopsys, Inc. 2025. ASIP Designer. https://www.synopsys.com/dw/ipdir.php?ds=asip-designer

[18] Blaise Tine, Varun Saxena, Santosh Srivatsan, Joshua R. Simpson, Fadi Alzammar, Liam Cooper, and Hyesoon Kim. 2023. Skybox: Open-Source Graphic Rendering on Programmable RISC-V GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, Vancouver, BC, Canada, 616–630. doi:10.1145/3582016.3582024

[19] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Hyesoon Kim. 2021. Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. Association for Computing Machinery, Athens, Greece, 754–766. doi:10.1145/3466752.3480128

[20] David Trilla, John-David Wellman, Alper Buyuktosunoglu, and Pradip Bose. 2021. NOVIA: A Framework for Discovering Non-Conventional Inline Accelerators. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 507–521. doi:10.1145/3466752.3480094

[21] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley, Berkeley, CA. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[22] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 23 (Jan. 2021), 29 pages. doi:10.1145/3434304

[23] Youwei Xiao, Fan Cui, Zizhang Luo, Weijie Peng, and Yun Liang. 2025. Cayman: Custom Accelerator Generation with Control Flow and Data Access Optimization. In *Proceedings of the 62nd Annual ACM/IEEE Design Automation Conference (DAC '25)*. IEEE Press, San Francisco, California, United States, 1–7. doi:10.1109/DAC63849.2025.11132875

[24] Youwei Xiao, Zizhang Luo, Kexing Zhou, and Yun Liang. 2024. Cement: Streamlining FPGA Hardware Design with Cycle-Deterministic eHDL and Synthesis. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 211–222. doi:10.1145/3626202.3637561

[25] Youwei Xiao, Chenyun Yin, Yitian Sun, Yuyang Zou, and Yun Liang. 2026. Finding Reusable Instructions via E-Graph Anti-Unification. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, Pittsburgh PA USA, 749–763. doi:10.1145/3779212.3790162

[26] Youwei Xiao, Yuyang Zou, Yansong Xu, Yuhao Luo, Yitian Sun, Chenyun Yin, Ruifan Xu, Renze Chen, and Yun Liang. 2025. Invited Paper: APS: Open-Source Hardware-Software Co-Design Framework for Agile Processor Specialization. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, Munich, Germany, 1–9. doi:10.1109/ICCAD66269.2025.11240817

[27] Ruifan Xu, Youwei Xiao, Jin Luo, and Yun Liang. 2022. HECTOR: A Multi-Level Intermediate Representation for Hardware Synthesis Methodologies. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22)*. Association for Computing Machinery, San Diego, CA, USA, 1–9. doi:10.1145/3508352.3549370

[28] Ruifan Xu, Yuyang Zou, and Yun Liang. 2026. FESTAL: Dataflow Accelerator Synthesis Framework with Graph-Based Fusion for FPGA. In *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, Hong Kong, China, 503–511. ISSN: 2153-697X. doi:10.1109/ASP-DAC66049.2026.11420389

[29] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: a scalable high-level synthesis framework with multi-level transformations and optimizations: invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22)*. Association for Computing Machinery, San Francisco, CA, USA, 1355–1358. doi:10.1145/3489517.3530631

[30] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proceedings of the ACM on Programming Languages* 7, PLDI, Article 125 (June 2023), 25 pages. doi:10.1145/3591239

[31] Jerry Zhao, Daniel Grubb, Miles Rusch, Tianrui Wei, Kevin Anderson, Borivoje Nikolic, and Krste Asanović. 2024. *The Saturn Microarchitecture Manual*. Technical Report UCB/EECS-2024-215. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-215.html

[32] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanović. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*. CARRV, Virtual Conference, 1–7. https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf