Finding Reusable Instructions via E-Graph Anti-Unification

Youwei Xiao
shallwe@pku.edu.cn
School of Integrated Circuits, Peking University
Beijing, China

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

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

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

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

Abstract

Domain-specific accelerators provide an increasingly valuable source of performance for diverse applications. Custom instructions that trigger the execution of dedicated hardware units or accelerators for common application functions become key building blocks in modern computing systems, balancing performance and cost effectiveness. RISC-V, the open and extensible instruction set architecture, is increasingly popularizing this trend. However, exploring custom instructions for an application domain remains challenging. Existing automated approaches suffer from poor reusability and limited performance. They can only identify or merge syntactically similar, scalar instruction sequences while missing semantically equivalent patterns.

We present ISAMORE, an end-to-end framework for discovering reusable custom instructions from domain applications. ISAMORE encodes general applications in an e-graph by constructing a structured domain-specific language. Its core methodology, *reusable instruction identification* (RII), leverages e-graph anti-unification (AU) to identify semantically equivalent common patterns across diverse applications, fully unleashing the potential of custom instructions. RII employs a phase-oriented iterative process with smart heuristics to enhance the scalability when dealing with real-world codebases. Besides, RII introduces the novel pattern vectorization technique, packing common operations from scalar programs into lanes of vectorized custom instructions to exploit data-level parallelism. Moreover, RII's Pareto-optimal pattern selection balances performance gains with area overheads, guided by a profiling-based hardware-aware cost model. Evaluation demonstrates ISAMORE's substantial performance gains, 1.12×-2.69×, over baseline approaches. We also demonstrate ISAMORE's practical potentials using various case studies, including library analysis for three application domains and hardware specialization for quantized LLM inference and post-quantum cryptography.

*Corresponding author.

1 Introduction

The relentless advancement of modern computational domains—such as digital signal processing [28, 60] and artificial intelligence [18, 47, 72]—has outpaced the capabilities of general-purpose processors. These fields demand computing power to meet the constraints of resource-limited systems like IoT devices and edge platforms. Custom instructions (CIs) coupled with specialized hardware units or accelerators have emerged as a powerful solution. The rise of the open and modular instruction set architecture RISC-V has further democratized this trend, enabling the wide adoption of various application customizations. Custom instructions can be flexibly added to the open-source RISC-V processors [7] and system-on-chips (SoCs) [6] for domain-specific acceleration. From the perspective of software, they can be invoked as naturally as native instructions and adopted seamlessly by existing compilers, preserving compatibility with established RISC-V software stacks to avoid nonnegligible software adaptation costs.

Designing these custom instructions, however, is a complex endeavor requiring expertise across both software and hardware domains. On the software side, developers must analyze workloads to identify performance bottlenecks and uncover *common execution patterns* that deserve acceleration. On the hardware side, the challenge lies in crafting accelerators and ensuring their smooth integration into the processor implementation. Application-specific instruction-set processor (ASIP) design tools [23, 50, 57, 69, 73] and accelerator design languages [16, 48, 49, 68, 70] ease some burdens by automating hardware generation, but they still leave the software-side efforts to human experts. Instruction customization frameworks [21, 29, 31, 53, 59, 67] aim to bridge this gap by automating the entire process from pattern identification to hardware implementation. Despite their promise, prior approaches suffer from a critical shortcoming: *low reusability*. Specifically, custom instructions occupy valuable core areas, making it essential to reuse them across different parts of an application or even multiple workloads.

However, existing methodologies concentrate on program *hotspots*, identifying program segments that execute repeatedly without considering *reusability*. Still, they only support *syntactic merging* [32, 59, 63] of similar patterns, but overlook *semantic equivalence* for full reusability. Our study shows that for *CImg* [25], the syntactic merging approach generates a large, over-specialized custom instruction, which is only used by 8 spots of the profiled applications. However, by considering semantic reusability, every custom instruction can accelerate 93 spots on average, achieving 1.17× more speedup with 90.5% area saving.

We propose a novel methodology, *reusable instruction identification (*RII*)*, to identify reusable custom instructions. The core idea of RII is to adopt *e-graph anti-unification (AU)* to generalize common substructures from the representative applications into reusable *patterns*. The rationale is that the e-graph data structure compactly represents sets of equivalent terms from the encoded application programs. After applying *equality saturation (EqSat)* with equational rewrite rules, RII identifies common patterns by *anti-unifying* [34, 37] pairs of e-classes, which requires the identified patterns to occur at least *twice*, fundamentally considering reusability for custom instructions. In addition, common expressions performed on multiple *lanes* can also be considered as reuse opportunities from scalar programs, creating vectorized custom instructions of *data-level parallelism (DLP)*. Consequently, the RII methodology innovatively adopts e-graph to identify reusable and parallel custom instructions.

We introduce the ISAMORE framework, which implements RII with novel algorithms and strategies. Applying e-graph AU for custom instruction identification faces many challenges. The first challenge is *how to represent application programs with casual control flow branching in an e-graph*. ISAMORE introduces a structured domain-specific language (DSL) to encode programs as e-graph terms, providing design entries for e-graph AU. The second challenge is *how to scale the e-graph AU for real-world applications*. The scalability issues are attributed to EqSat's exploding e-graph size and e-graph AU's pattern identification complexity, both of which grow exponentially to become intractable. RII introduces a *phase*-oriented iterative process for pattern identification, which applies partial rewrite rules in each phase for EqSat to control the e-graph scale. The iterative flow also identifies patterns over previously identified ones, leading to more reusable patterns. To overcome e-graph AU's exponential complexity, RII introduces *smart AU* techniques. It conducts structural hashing and term typing on the e-graph to filter e-class pairs of high similarity and consistent term types, instead of exhaustive traversals on all e-graph pairs. It also introduces heuristic *pattern sampling* strategies to reduce the design space to only useful patterns.

For the last challenge, *how to identify custom instructions for ideal performance gains*, RII introduces a hardware-aware cost model and a multi-objective pattern selection process to explore tradeoffs between performance gains and hardware overheads of the synthesized custom instructions. RII further introduces the *pattern vectorization* technique, which identifies *seeds* through e-graph AU and produces vectorized custom instructions through e-graph operations, exploiting DLP potentials. To the best of our knowledge, ISAMORE is the first end-to-end framework to customize reusable and parallel instructions from real-world applications through scalable e-graph anti-unification.

This work offers the following contributions:

- We introduce *reusable instruction identification (*RII*)*, the first practical methodology that identifies reusable custom instructions through e-graph anti-unification.
- We propose a phase-oriented iterative process with *smart AU*, *pattern vectorization*, and *hardware-aware selection* techniques to improve tractability and instruction quality.
- We present an open-source[1], end-to-end framework, ISAMORE, that implements RII for systematic custom instruction identification from real-world domain applications.

Evaluation. We evaluate ISAMORE on nine benchmark kernels and one compound benchmark comprised of all the kernels, showing that ISAMORE outperforms two reusability-unaware instruction customization baselines, including a fine-grained enumeration approach combining [22, 29] and the coarse-grained NOVIA [59], by up to 2.69×. We also conduct rich case studies for evaluation. We run ISAMORE on open-source libraries for three application domains, showing 1.17×-2.73× speedup against NOVIA and demonstrating ISAMORE's scalability. We further run ISAMORE to generate RoCC [7] accelerators for both quantized large language model (LLM) inference and post-quantum cryptography (PQC) algorithm, running RTL simulation and physical implementation for precise evaluation. The case studies demonstrate ISAMORE's practical effectiveness in identifying and implementing custom instructions for real-world applications.

2 Background and Motivation

This section presents the background of instruction customization and the potential of e-graph-based methodologies as a solution for finding reusable instructions.

2.1 Instruction Customization Landscape

As summarized in Table 1, prior custom instruction design approaches fall into three main categories. Manual description-based approaches, supported by commercial [12, 23, 57] and academic [50] tools, empower expert designers to specify custom instructions using architecture description languages (ADLs). These tools offer high flexibility but are inherently non-automatic, requiring significant manual effort. In contrast, automated approaches identify custom instructions

[1] https://github.com/pku-liang/ISAMORE

Table 1. Comparison of instruction customization works

Approach	Auto--matic?	Reuse-aware	Merging strategy	Granularity	Vectori--zation?
Description [12, 23, 50, 57]	✘	/	/	SG BB HB CF	✔
Fine-grained [21, 22, 29, 53]	✔	✘	syntactic	SG BB HB CF	✘
Coarse-grained [31, 32, 59]	✔	✘	syntactic	SG BB HB CF	✘
ISAMORE (This work)	✔	✔	semantic	SG BB HB CF	✔

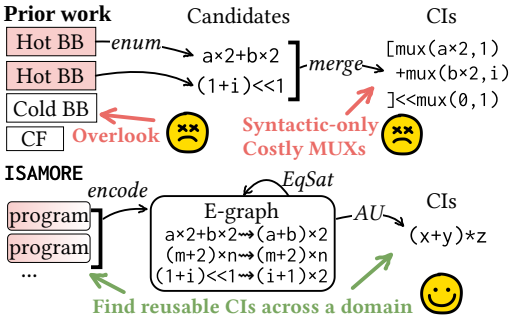


Figure 1. Comparison example of instruction customization.

directly from application code. Early works explored *fine-grained* automation [21, 22, 53], enumerating small, convex subgraphs within basic blocks. Their limited scope often misses larger, more impactful optimization opportunities. *Coarse-grained* methods [31, 32, 59] merge code scopes, such as basic blocks, to create larger custom instructions.

Despite these advancements, significant limitations remain. A primary issue is *low reusability*. Automated methods, whether fine-grained or coarse-grained, rely on *syntactic analysis* of program hotspots, overlooking the *semantic equivalence* that could enable a single custom instruction to accelerate multiple, syntactically different code segments across the whole application. This leads to over-specialized custom instructions with poor utilization. Figure 1 shows an example of over-specialization. Existing methods only consider two expressions from the hotspot basic blocks, `a×2+b×2` and `(1+i)«1`, and syntactically merge them to generate the custom instruction of four operations and three MUXs, which is hard to reuse and inefficient in hardware implementation. Instead, our semantic-aware approach identifies a concise and reusable custom instruction of two operations.

This frequently happens in real-world applications. We analyzed three open-source libraries as detailed in Section 7.2.1, and found that syntactic merging generates inefficient custom instructions that are reused by only 6.4-9.0 times. In comparison, considering semantic reusability during identification, the average reuse factor per instruction rises to 19.1-93.0, achieving an average speedup of 1.17×-1.64× with

$$
\begin{aligned}
&\text{E-graph} && \mathcal{G} = \langle C, M \rangle,\ M\colon \mathcal{I} \to C \\
&\text{E-class ids} && a, b \in \mathcal{I} \\
&\text{E-nodes} && n ::= s(a_1, \ldots, a_k) \in N,\ s \in \Sigma \\
&\text{E-classes} && c ::= \{n_1, \ldots, n_m\} \in C \\
&\text{Rewrite rule} && l \rightsquigarrow r, \text{ where } l, r \in \mathcal{T}(\Sigma, \mathcal{X}) \\
&\text{E-class AU} && AU(a,b) = \bigcup_{n_a \in M(a), n_b \in M(b)} AU(n_a, n_b) \\
&\text{E-node AU} && AU(s(a_1, \ldots, a_k), s(b_1, \ldots, b_k)) \\
& && \quad = \{s(p_1, \ldots, p_k) \mid p_i \in AU(a_i, b_i)\} \quad (0) \\
& && AU(s_1(\ldots), s_2(\ldots)) = ?x \text{ , if } s_1 \neq s_2
\end{aligned}
$$

Figure 2. Formulas for e-graph anti-unification. Σ denotes a set of *constructors*. $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of *patterns* over Σ and $\mathcal{X}$, the set of *variables*. $?x$ denotes a new variable.

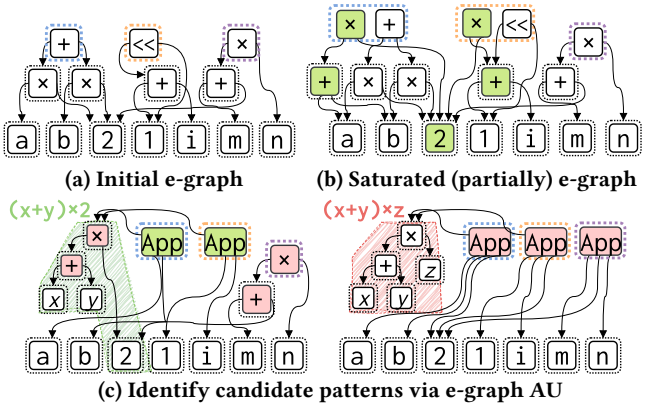


Figure 3. E-graph and anti-unification example.

84.0%-93.2% area saving. Another critical flaw is *restricted parallelism*. Table 1 shows that prior works overlook data-level parallelism (DLP) to generate vectorized custom instructions. We studied a set of benchmark kernels as detailed in Section 7.1, and found that exploiting DLP further improves performance by up to 1.68× compared to scalar custom instructions. These remarkable potentials for improvement motivate us to explore an automated, semantic-aware methodology to identify reusable and parallel custom instructions.

2.2 E-graph Anti-Unification

To overcome the limitations of syntactic analysis, our work leverages a powerful combination of the e-graph [45] and anti-unification [13]. An *e-graph* is a data structure that compactly represents semantically equivalent terms. As formally defined in Figure 2, an e-graph consists of *e-classes*, which group together equivalent *e-nodes*. Each e-node represents a function symbol (also referred to as *constructor* in this paper)

applied to argument e-class identifiers. For instance, Figure 3a shows an initial e-graph that contains the expressions in Figure 1, such as a×2+b×2, with each dashed rectangle denoting an e-class. The true power of an e-graph is unlocked through *equality saturation* (EqSat), a process that exhaustively applies rewrites from the given *ruleset* to find all equivalent terms. During the process, each rewrite rule searches its left-hand side (LHS) pattern within the e-graph, and for every match, instantiates a new e-node (if not existent) according to the right-hand side (RHS) pattern and unions e-classes. Figure 3b illustrates this: applying the rule ?x×c+?y×c ⇝ (?x+?y)×c creates two e-nodes for the term (a+b)×2, with the root merged into the blue dotted e-class.

While EqSat reveals equivalence, *anti-unification (AU)* reveals commonalities. AU is a generalization technique that computes the *least general generalization* [34, 37] of two terms, creating a template that captures their common structure. For example, anti-unifying (a+b)×2 and (1+i)×2 yields (?x+?y)×2. Building on this, Cao et al. [13] proposes the *library learning modulo theories* (LLMT) algorithm, which runs EqSat and performs AU on the saturated e-graph to generalize the common patterns. As formalized in Figure 2, e-graph anti-unification recursively traverses pairs of e-classes and e-nodes to find sets of common patterns. As shown in Figure 3c, by anti-unifying the e-classes for (a+b)×2 and (1+i)«1, the algorithm can identify a pattern (?x+?y)×2. This ability to find common patterns among semantically equivalent terms is the key insight of our work.

LLMT cannot be directly applied to the custom instruction identification task due to the following challenges. First, it cannot take general programs with unstructured control flow as input, which cannot be represented as e-graph terms. Second, the scalability of the e-graph size and the number of AU patterns grows exponentially, making the process intractable. Specifically, it exhaustively enumerates every pair of e-classes, and, for each pair, traverses their term structures to generate all possible anti-unifiers as candidate patterns, whose number grows exponentially due to the Cartesian product of child e-class AUs for every e-node pair, as shown in Figure 2. In our evaluation, LLMT failed to complete most cases with >150 e-classes under practical resource limits (30GB memory), while most real-world applications exceed 2000 e-classes. Third, it cannot achieve ideal performance gains since it does not fully exploit parallelism opportunities and use program size minimization as the optimization objective, which misaligns with the instruction customization task. These challenges necessitate a fundamental rethinking of e-graph anti-unification for custom instruction identification, motivating this work.

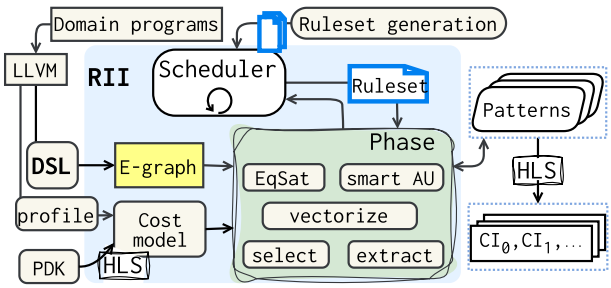


Figure 4. Overview of the ISAMORE framework.

3 Framework Overview

As illustrated in Figure 4, ISAMORE is an end-to-end framework that begins with a set of domain-representative programs and finally generates sets of custom instructions. The compiled LLVM IR is translated into ISAMORE's structured DSL to be encoded as an e-graph, which serves as the design entry of the subsequent flow (Section 4). RII, the core methodology of ISAMORE, is a phase-oriented iterative process for identifying reusable patterns (Section 5.1). The offline ruleset generation process constructs the rulesets for each RII phase to apply. RII innovatively introduces the *smart AU* technique for scalable e-graph anti-unification (Section 5.2), and proposes *pattern vectorization* to exploit DLP opportunities (Section 5.3). The phase's pattern selection and program extraction decisions are guided by a tangible hardware-aware cost model (Section 5.4), which leverages the realistic performance information collected by instrumenting and profiling the LLVM IR, and uses a high-level synthesis (HLS) engine for hardware preformance and overhead estimation towards the target PDK. Finally, RII solutions, sets of reusable patterns, are translated into sets of custom instructions with hardware implementation through the HLS engine.

4 Structured DSL

At the entry point of the RII workflow is a domain-specific language (DSL) designed to represent general programs as e-graph terms. It captures program semantics, including control flows, in a structured, dataflow-centric manner.

4.1 DSL Syntax

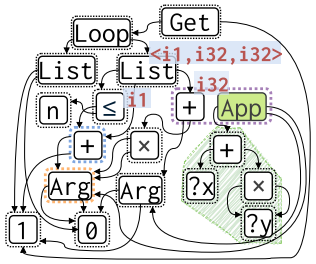
The syntax of ISAMORE's structured DSL is defined in Figure 5. It includes arithmetic, logical, and memory access operations. We introduce two operations, Loop and If, to represent program control flows. Especially, a Loop operation represents a do-while loop. The first argument, e_{in}, passes the initial values of the loop-carried variables to the loop body, and the second argument, e_{body}, represents the loop body with loop condition and loop-carried variables combined as the output list. Besides, the structured DSL also includes vector operations, including Vec that creates a vector from scalar values, and the vector version of the

$?x \in$	PatVars	(Pattern Variables, *Holes*)
$uop \in$	{Neg, Not, ...}	(Unary Ops)
$bop \in$	{Add, Load, ...}	(Binary Ops)
$top \in$	{Mux, Store, ...}	(Ternary Ops)
$e ::=$	$l \in \mathbb{Z} \cup \mathbb{F}$	(Literal)
	$\mid uop(e)$	(Unary Op)
	$\mid bop(e_1, e_2)$	(Binary Op)
	$\mid top(e_1, e_2, e_3)$	(Ternary Op)
	$\mid$ Arg(id, i)	(Argument)
	$\mid$ If$(e_{\text{in}}, e_{\text{then}}, e_{\text{else}})$	(Conditional)
	$\mid$ Loop$(e_{\text{in}}, e_{\text{body}})$	(Loop)
	$\mid$ List$(e_1, \ldots)$	(Create List)
	$\mid$ Get$(e_{\text{list/vec}}, i)$	(Get from List/Vector)
	$\mid$ Vec$(e_1, \ldots)$	(Create Vector)
	$\mid$ Vec $uop(e_{\text{vec}})$	(Vector Unary Op)
	$\mid$ Vec $bop(e_{\text{vec}_1}, e_{\text{vec}_2})$	(Vector Binary Op)
	$\mid$ Vec $top(e_{\text{vec}_1}, e_{\text{vec}_2}, e_{\text{vec}_3})$	(Vector Ternary Op)
	$\mid ?x$	(Pattern Variable)
	$\mid$ App(e_{pat}, e^*)	(Apply Pattern)

Figure 5. Formal syntax of ISAMORE's structured DSL.

```
Get(Loop(List(1,0),
 List(
  Le(Add(Arg(0,0),1),n),
  Add(Arg(0,0),1),
  App(
   Add(?x,Mul(?y,?y)),
   Arg(0,1),Arg(0,0))
 )),1)
```

(a) Structured DSL



(b) E-graph representation

Figure 6. Square sum function represented in ISAMORE.

unary, binary, and ternary operations. Finally, it introduces pattern-related constructs for custom instruction identification, including pattern variables (e.g., $?x$) to specify the arguments of a candidate pattern, and an App operation to apply a candidate pattern to arguments.

4.2 E-Graph Encoding

ISAMORE's structured DSL represents general programs in the e-graph data structure. For every expression e in the DSL, it directly maps to an e-node whose *constructor* is e's operation name and whose children are the e-classes containing the arguments of e. Figure 6a shows the DSL representation of the square sum ($\sum_{i=1}^{n} i^2$) function, which constructs the e-graph in Figure 6b. The reduplicated expressions in the DSL, such as Add(Arg(0,0),1), are absorbed into the same e-class in the e-graph for compactness. In addition to representing original programs, identified patterns and their applications are also encoded in the e-graph through the App operation in the DSL.

```
1  # Structured DSL dsl, rulesets S_{R≡}, profiling p
2  def RII(dsl, S_{R≡}, p):
3    G = egraph(dsl)
4    F_P = pareto(∅)
5    S = scheduler(S_{R≡}, p)
6    while R≡ = S.next(F_P):
7      P_pre = ⋃_{P∈F_P} P
8      G_init = vectorize(G) if isVector(R≡) else G
9      G' = eqSat(G_init, κ(P_pre) ∪ R≡)
10     P_cand = smartAU(G')
11     F = select(G', P_cand, p)
12     foreach P in F:
13       update(F_P, extract(G', κ(P), p))
14   return F_P
```

Figure 7. The top-level RII algorithm.

4.3 Type System

ISAMORE's structured DSL is strongly typed. It utilizes *e-class analysis* [45] to infer the result type for every e-class, building a mapping from e-classes to the *result type* domain. While type propagation for standard dataflow operations follows standard algebraic rules, the control flow constructs enforce specific structural constraints defined over type tuples. Formally, let $\langle \tau_1, \ldots, \tau_k \rangle$ denote a tuple of data types and i1 denote a boolean control signal. For If$(e_{\text{in}}, e_{\text{then}}, e_{\text{else}})$, the input e_{in} must aggregate the condition and arguments as type $\langle \text{i1}, \tau_1, \ldots, \tau_k \rangle$, with both branches, e_{then} and e_{else}, taking the same arguments and yielding values of the consistent result type $\langle \tau'_1, \ldots, \tau'_{k'} \rangle$. In contrast, for Loop$(e_{\text{in}}, e_{\text{body}})$, the input e_{in} initializes loop-carried values of type $\langle \tau_1, \ldots, \tau_k \rangle$, while the body e_{body} returns $\langle \text{i1}, \tau_1, \ldots, \tau_k \rangle$ to capture both the loop-continuation condition and the updated looped-carried values. The type system is utilized by the RII workflow to improve the efficiency of the AU process, as discussed in Section 5.2.

5 Reusable Instruction Identification

This section introduces RII, the methodology for identifying custom instructions of ideal reusability and parallelism through tractable e-graph anti-unification.

5.1 Phase-Oriented Iteration

Figure 7 presents the overall algorithm workflow of RII. It initializes the egraph from the structured DSL (line 3) and creates an empty Pareto front $\mathcal{F}_\mathcal{P}$ (line 4) to hold solutions, each of which is a set of patterns for performance-overhead tradeoff. The central process of RII is applying phase-oriented iterations for pattern identification and selection (lines 6-13). The key insight is to abandon the single, monolithic application of all rewrite rules. Instead, RII runs a sequence of phases, where each phase applies a smaller, carefully selected

Structural Hashing

Domain $D_{\mathcal{H}} \subset \{0,1\}^{64}, sh : (\mathcal{I} \cup \mathcal{N}) \to D_{\mathcal{H}}$

E-class $sh(a) = \left((\sum_{n \in a} sh(n)_i) > \frac{\text{size}(a)}{2} \right)_{i=0}^{63}$

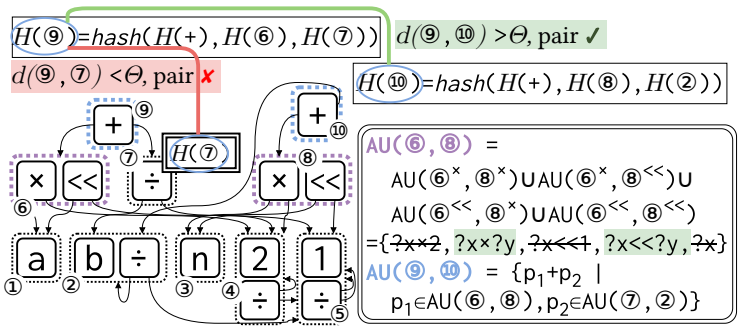
E-node $sh\,(s(a_1, \ldots, a_k)) = \text{Bits}_{64}\left(\text{Hash}_{u64}(sh(s), sh(a_1), \ldots, sh(a_k))\right)$

Similarity: $d(a,b) = \sum_{i=0}^{63} (sh(a)_i \neq sh(b)_i)$

Sampling Feature: $f : \mathcal{T}(\Sigma, \mathcal{X}) \to \mathbb{R}$

$$\mathbf{F}\,(s(p_1, \ldots, p_k)) = [f(p_1), \ldots, f(p_k)]$$

(a) Formulas for structural hashing and sampling features.



(b) Example for terms `{a×2+b÷2, (n«1)+b}`.

Figure 8. Smart AU identification: (a) definitions and (b) an example of e-class pairing and AU pattern sampling.

ruleset, denoted as $\mathcal{R}_{\equiv}$ at line 6. Specifically, $\mathcal{P}_{pre}$ at line 7 denotes the union of all pattern sets explored by previous iterations. RII vectorizes the e-graph if required (line 8), conducts EqSat with both pattern-application rewrites $\kappa(\mathcal{P}_{pre})$ and rewrites from the ruleset $\mathcal{R}_{\equiv}$ (line 9), and then identifies and selects new patterns to update $\mathcal{F}_{\mathcal{P}}$ (line 10-13).

The iterative approach's advantages are two-fold. First, it controls the e-graph scale by applying smaller rulesets, avoiding the unmanageable e-graph size. Second, it enables the progressive discovery of increasingly reusable patterns. Specifically, RII identifies common substructures of patterns from previous phases through the rewrites $\kappa(\mathcal{P}_{pre})$. Figure 3c shows an example, where `(x+y)×2` is generalized into `(x+y)×z` in a subsequent phase.

Ruleset Construction. The effectiveness of this approach hinges on ruleset selection. We classify rewrite rules along several orthogonal dimensions into *base rulesets* at offline. Considering the impact on the e-graph's size, we classify rules as *sat* or *nonsat*, where *sat*, shortly for *saturating*, indicates that the rewrites won't introduce new e-classes. Besides, based on the variable types of rewrites, we classify rules into *int* and *float*. Moreover, we classify rules into *vector* and *scalar* according to the existence of vector terms. Base rulesets are not disjoint, and we select their union or intersection to flexibly create rulesets for each phase.

Phase Scheduling. The iterative process is controlled by a *phase scheduler*, notated as S in Figure 7. It decides the ruleset to use and the termination of the iteration. Specifically, RII's phase scheduler applies *int-sat* and *float-sat* rulesets in the first two phases, respectively, both of which saturate the e-graph. Then, for every subsequent phase, the scheduler selects n rules from the *nonsat* rulesets and applies each selected rule *twice*. The iteration stops when the solution set $\mathcal{F}_{\mathcal{P}}$ remains unchanged. The phase scheduling strategy considers both the equivalence exploration and the e-graph growth. In early phases, the equivalence behind the *sat* results is fully exposed without boosting the e-graph's scale, which provides reuse opportunities for pattern identification as much as possible. Then in the following phases, our strategy also exploits the equivalence behind the *nonsat* rulesets, but applies the rules in a selective and restrained manner to avoid e-graph explosion. In this way, the strategy reaches a good balance between the solution quality and the algorithm efficiency of the RII workflow.

This strategy not only fully exploits the equivalence behind the *sat* rulesets through the complete saturation, but also covers non-saturating rules with a limited number of uses to avoid e-graph explosion.

5.2 Smart AU Identification

RII introduces the *smart AU* technique to address the scalability issue of e-graph AU. It comprises two key heuristics:

Similarity-based e-class pairing. RII heuristically selects only e-class pairs likely to yield meaningful patterns based on e-class similarity. We use *e-class analysis* [45] to associate metadata from a defined domain to each e-class and propagate it for congruence closure. In addition to the result type domain described in Section 4.3 for pairing type-consistent e-classes, RII further introduces the *structural hashing* domain, $D_{\mathcal{H}}$, to pair structurally similar e-classes. Specifically, e-class pairs of different result types are excluded from pairing. Figure 8a presents the defination. We maintain a 64-bit structural hash *sh* for each e-class and e-node. The propagation of *sh* follows a recursive scheme: every e-node applies a 64-bit hash function on its constructor and its children's *sh* values, while every e-class aggregates the contained e-nodes' *sh* values via majority voting at each bit position. Specifically, the structural hashing assigns uniform *sh* values for different literals, arguments, and pattern variables, eliminating their influence on the e-class paring for better structural matching used to identify common patterns. RII then quantifies structural similarity between two e-classes using the Jaccard distance. Only pairs with a similarity score above a predefined threshold Θ will be explored. Figure 8b shows an example, where the e-class ⑨ and ⑩ are structurally similar to be paired, and the e-class ⑨ and ⑦ are

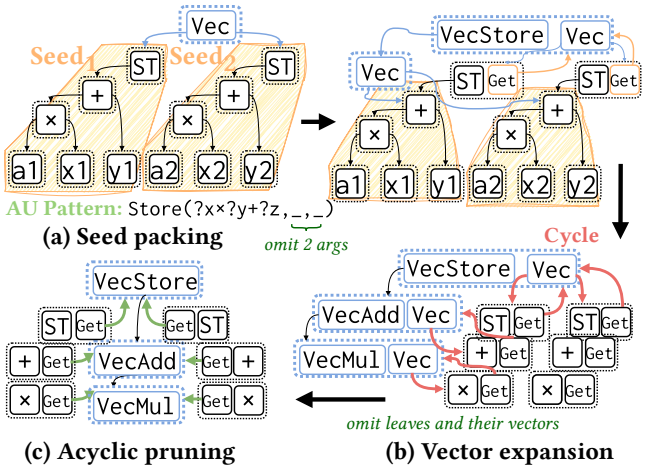


Figure 9. Example of pattern vectorization

not. Combining the result type domain and the structural hashing domain, RII greatly reduces the number of e-class pairs for exploration, improving the effectiveness of the AU process.

Heuristic AU pattern sampling. RII samples a representative subset of patterns for e-node *AU* sets to avoid the exponential growth. Specifically, RII employs a feature function f that maps each pattern to a real value, considering both the latency (prioritized) and area (secondarily). RII introduces two sampling heuristics, *boundary* and *kd-tree*. The *boundary* strategy only keeps *extreme* patterns that either minimize or maximize the feature value, formalized as:

$$AU_{\mathcal{B}}\,(s(a_1,\ldots,a_k),s(b_1,\ldots,b_k)) = \{\arg\min_{p\in\mathcal{P}_{\mathcal{B}}} f(p), \arg\max_{p\in\mathcal{P}_{\mathcal{B}}} f(p)\}$$
$$\text{where, } \mathcal{P}_{\mathcal{B}} = \{s(p_1,\ldots,p_k) \mid p_i \in AU_{\mathcal{B}}(a_i,b_i)\}$$

This heuristic brutally selects *two* very different patterns, improving the algorithm's efficiency. Instead, the *kd-tree* strategy tends to select broader distinct patterns according to the feature vector $\mathbf{F}(p)$ for each pattern p, as formalized in Figure 8a. It builds a KD-tree [9] data structure for potential AU patterns $p \in \mathcal{P}_{kd}$ with $F(p)$ as coordinates, and partitions the overall space into $m = 2^d$ cells by the first d dimensions of the KD-tree. We then sort the patterns in each cell by the feature function f and sample β evenly spaced patterns from the sorted sequence. Finally, we union the sampled patterns to form the final AU_{kd} of $m\beta$ patterns. The *kd-tree* strategy mediates the algorithm efficiency for sampling coverage.

Combining similarity-based pairing with heuristic sampling, *smart AU* makes pattern identification in e-graphs both scalable and effective. As illustrated in Figure 8b, our approach processes structurally similar e-class pairs (e.g., ⑨ and ⑩) and keeps representative patterns per pair (two patterns for ⑥ and ⑧), guiding identification to be tractable.

5.3 AU-based Pattern Vectorization

RII exploits DLP from scalar inputs based on the key insight: a vectorized pattern can be conceptualized as one pattern applied across *lanes* of terms, and e-graph AU can identify lane-level common patterns. RII introduces the novel *pattern vectorization* technique, comprising the following steps:

Seed packing: locates the vectorization candidates-the *seeds*-within the scalar e-graph and transforms the e-graph to contain hybrid scalar-vector terms. We first run *smart AU identification* to find patterns that match multiple e-graph terms. For each identified scalar pattern, we use *e-matching* to find instances (*seeds*), whose roots form a *seed pack* if the instance terms belong to the same basic block in the original program. We then insert Vec e-nodes to unify the e-classes of a seed pack as a vector term, as illustrated in Figure 9a.

Pack expansion: constructs hybrid scalar-vector e-graphs. It is unrealistic to learn vectorized patterns without vectorized constructors in the e-graph. Therefore, we use *lift* rewrites from the *vector* ruleset to recover the vector constructors by expanding the seed packs. A *lift* rewrite finds Vec applied on terms of the same scalar constructor, and replaces every match with a vectorized equivalent, such as the rule for introducing the vectorized add constructor: `(Vec (Add a b) (Add c d))⇝(VecAdd (Vec a c) (Vec b d))`. To enable data flow from vector to scalar in the e-graph, we also apply *couple* rewrites, which insert Get e-nodes to extract scalar lanes from a Vec term. Figure 9b shows an example of the expansion process, preparing a hybrid scalar-vector e-graph for vectorized pattern identification.

Acyclic pruning: reduces the e-graph scale with cycles eliminated. The e-graph tightly coupling scalar and vector terms exhibits two critical limitations: e-graph scale explosion due to combinational length-agnostic packing decisions among seeds, and the existence of Get→Vec→Get cycles as illustrated in Figure 9b, which severely degrades the extraction quality (Section 5.4), even leading to infeasible cases. To address the issues, we first employ a greedy e-graph extractor with a custom cost function that deliberately favors vector constructors of high DLP. The extracted terms represent a specific vectorization scheme, which is then fused into the original scalar e-graph through a variant of Enumo [51]'s *compress* operation. This approach keeps a specific vectorization scheme in the e-graph rather than storing all vectorization opportunities, which avoids duplicated packing and cycles. Although the pruning heuristic might miss opportunities for vectorization and instruction identification due to the greedy nature of the extractor, it effectively reduces the e-graph's scale and eliminates any cycles, which is essential for both AU and extraction. In practice, it keeps non-overlapping vectorization operations of high DLP as much as possible, enabling AU-based pattern identification

from them. Eventually, the acyclic pruning yields a lightweight and acyclic e-graph that maintains tight scalar-vector coupling, as illustrated in Figure 9c.

The hybrid scalar-vector e-graph is then used for subsequent phases of pattern identification and selection, as shown in Figure 7, comprehensively considering vectorized and scalar candidate patterns for overall performance.

5.4 Hardware-Aware Selection and Extraction

RII introduces a profiling-based, hardware-aware cost model to guide the multi-objective pattern selection and extraction.

5.4.1 Cost model: estimates the performance and area impact of introducing an identified pattern as a custom instruction, as formalized:

$$\Delta_L(p) = \sum_{u \in use(p)} \left(\sum_{o \in p} \mathrm{CPO}\left(bb(o,u)\right) - L_{HLS}(p) \right) \quad (1)$$

$$S(\mathcal{P}) = \frac{L_{\mathrm{cpu}}}{L(\mathcal{P})} = \frac{L_{\mathrm{cpu}}}{L_{\mathrm{cpu}} - \sum_{p \in \mathcal{P}} \Delta_L(p)} \quad (2)$$

$$A(\mathcal{P}) = \sum_{p \in \mathcal{P}} A_{HLS}(p) \quad (3)$$

where $\Delta_L(p)$ denotes the total latency saving of pattern p reused multiple times over the software execution. For each use u, the latency saving is calculated by subtracting the hardware accelerator latency from the software latency. Software latency is estimated as the sum of the profiled cycles per operation (CPO) of the basic block $bb(o,u)$ for each constructor o in p. Hardware latency is reported by a lightweight high-level synthesis (HLS) engine performing as-soon-as-possible scheduling for p's serialized behavior. Especially, the HLS engine applies loop pipelining when p contains Loop constructors. Notably, latencies are calculated in nanoseconds considering the device frequency. The overall speedup and the area overhead are calculated according to Eq 2 and Eq 3, respectively.

5.4.2 Multi-objective selection. We leverage *e-class analysis* [45] to find a set of Pareto-optimal solutions, where each solution represents a trade-off between speedup and area. The analysis associates a Pareto front $\mathcal{F}$ for each e-class and e-node, whose element $\mathcal{P}_k$ is a pattern set. The propogation of $\mathcal{F}$ is formalized as:

$$\mathcal{F}(s(a_1, \ldots, a_k)) = \prod \mathcal{F}(a_i) = \{\bigcup \mathcal{P}_i \mid \mathcal{P}_i \in \mathcal{F}(a_i)\}$$
$$\mathcal{F}(\mathtt{App}(p, a_1, \ldots, a_k)) = \{\mathcal{P} \cup \{p\} \mid \mathcal{P} \in \prod \mathcal{F}(a_i)\}$$
$$\mathcal{F}(a) = \mathrm{prune}\left(\bigcup_{n \in M(a)} \mathcal{F}(n)\right)$$

For a non-App e-node, the $\mathcal{F}$ is calculated by constructing the Cartesian product of its children's $\mathcal{F}$s. For an App e-node, each pattern set in the product is extended by the pattern applied. The Pareto optimality is maintained during the propagation, with the latency saving (Eq 1) approximated by substituting pattern p's actual uses by its matches in the e-graph. For an e-class, it unions the $\mathcal{F}$s of its e-nodes, and conducts the prune operation to keep only the top K solutions considering the prioritized speedup metric in a *beam search* manner. The final Pareto front of the root e-class provides the selection solutions.

5.4.3 Extraction and refinement. To derive more accurate cost modeling and truly promising solutions, we perform a final extraction and refinement step on the selected solutions. For each non-dominated solution $\mathcal{P}$, we perform e-graph *extraction* with the patterns applied as rewrite rules $\kappa(\mathcal{P})$. The extraction cost function is configured to maximize total latency saving calculated by Eq 1 without approximation. After extraction, we perform *fidelity refinement*, which recalculates the cost model (Eq 2 and Eq 3) on the extracted program with the patterns adopted, and updates the global solution set $\mathcal{F}_{\mathcal{P}}$, as shown in line 13 of Figure 7.

6 Implementation

ISAMORE is implemented as a comprehensive application analysis and instruction customization framework. The frontend implements two LLVM passes with LLVM 18 [39, 41]. The first transforms LLVM IR into ISAMORE's structured DSL, building upon JLM [56]'s implementation for control flow restructuring. The second pass instruments the LLVM bitcode, inserting markers at basic block boundaries. To acquire realistic performance data, we implemented a custom tracer in GEM5 [43] to detect the instrumented markers and collect detailed performance metrics, including cycles per operation (CPO) and execution count for each marked basic block, which informs the cost model in Section 5.4.1.

The core of ISAMORE materializes the entire RII workflow in Rust, building upon the egg [45] and babble [13] frameworks with 20157 LOC extension. For the cost model, ISAMORE's HLS engine calls the regression-based operation delay and area estimators from XLS [33]. For offline rewrite ruleset generation, we implement Enumo [51] traits for the DSL and describe rule enumeration with an SMT backend for equivalence checking (881 LOC). Finally, ISAMORE synthesizes the solution patterns into Verilog through CIRCT [20].

7 Evaluation

We evaluate ISAMORE by addressing the following questions:

1. Can RII techniques enable e-graph anti-unification for instruction identification? (Section 7.1.1)
2. How do ISAMORE-identified custom instructions perform compared to the baseline approaches? (Section 7.1.2)
3. How do RII's heuristics and features affect tractability and solution quality? (Section 7.1.3)
4. Can ISAMORE scale to real-world applications? (Section 7.2.1)
5. How can ISAMORE help designers to explore hardware specialization in practice? (Section 7.2.2, Section 7.2.3)

Methodology. We report the overall speedup calculated by Eq 2 for the performance study. The target clock frequency is set to 1GHz for HLS scheduling. For hardware overheads,

Table 2. Benchmark kernels and ISAMORE's running statistics with RII features enabled or not. The e-graph size, either original or peak, is reported by the number of e-nodes. $|\mathcal{P}_{cand}|$ denotes the number of identified candidate patterns.

Benchmark	Description	LLVM IR LOC	Orig. Size	Peak Size		$\lvert\mathcal{P}_{cand}\rvert$		Runtime		Memory	
				LLMT	RII	LLMT	RII	LLMT	RII	LLMT	RII
2DConv	2D convolution	169	339	10286	400	633K	133	~~707s~~	25s	>30GB	60MB
MatMul	Matrix multiply	94	189	10406	263	233K	49	~~516s~~	24s	>30GB	138MB
MatChain	Matrix chain multiplication	103	218	10377	294	19K	46	~~860s~~	41s	>30GB	187MB
FFT	Fast Fourier Transform	132	263	10505	272	299K	68	~~610s~~	16s	>30GB	39MB
Stencil	2D stencil	88	184	10225	260	16K	61	~~104s~~	41s	>30GB	100MB
QProd	Quaternion product	194	224	10293	333	80K	192	~~221s~~	98s	>30GB	231MB
QRDecomp	QR decomposition	496	1426	10319	1728	34K	127	~~2538s~~	145s	>30GB	799MB
Deriche	Deriche edge detector	270	677	10380	783	26K	77	~~3075s~~	27s	>30GB	126MB
SHA	SHA-256 secure hash algorithm	339	798	10944	987	94K	89	~~3580s~~	83s	>30GB	173MB

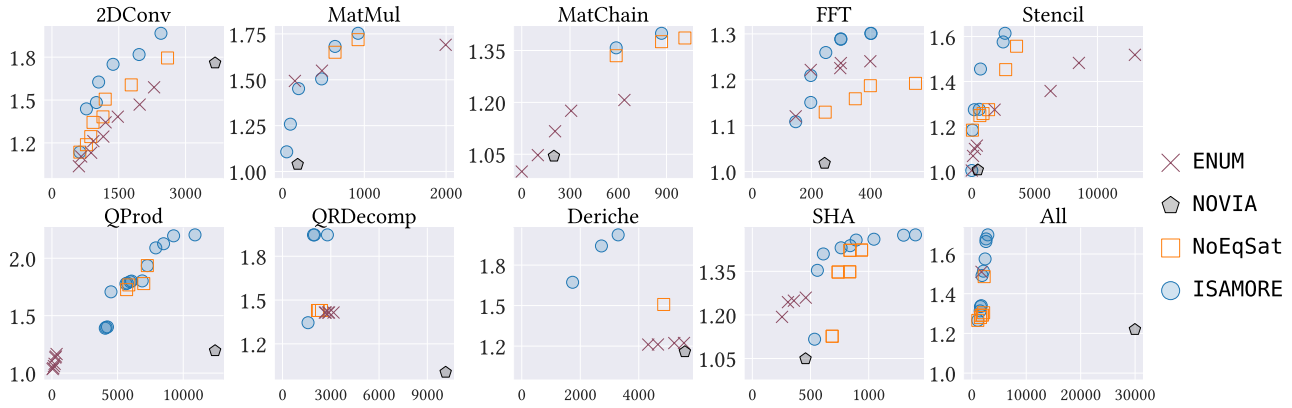


Figure 10. Performance gains and area overheads of Pareto solutions produced by ISAMORE and baseline approaches. The x-axis denotes the total area (μm^2), and the y-axis denotes the overall speedup (×) over general-processor execution.

we report the final synthesized area produced by the OpenROAD flow (v2.0-16235) [4] targeting ASAP7 PDK. We run evaluations on a machine equipped with two AMD EPYC 7763 CPUs (128 cores/256 threads, 1.8TiB RAM) running Ubuntu 22.04. For each run, we open a standalone process with a timeout of 2.5 hours and a memory limit of 30 GB by default. ISAMORE's offline ruleset generation introduces 1164 equational rewrite rules after 20 hours of enumeration.

7.1 Identifying Instructions for Benchmark Kernels

Table 2 shows the benchmark kernels used in our evaluation. They are a collection of kernels for use cases in computer vision, machine perception, digital signal processing, and cryptography. The source code are retrieved from Diospyros [60], PolyBench [42], Machsuite [54], and Coremark-PRO [1]. We count and report the LLVM IR LOC for the kernel functions, which range from 88 to 496 after LLVM's optimization with the flag `-O3`. LLVM conducts loop unrolling, exposing reuse and vectorization opportunities. For profiling, we set up the GEM5 simulation configuration based on the `O3_ARM_v7a_3` CPU model with 32KiB L1-I/L1-D and 1MiB L2 cache.

7.1.1 Comparison with vanilla e-graph AU..

Table 2 compares the running statistics for RII features disabled and enabled. With RII disabled, ISAMORE runs the vanilla LLMT [13] in a single-phase, exhaustive manner. We configure RII to adopt the *boundary* strategy for *pattern sampling*, which is denoted as the `Default` mode. The results show RII effectively reduces the peak e-graph sizes by 6-39×, thanks to RII's phase-oriented iterative approach. LLMT's candidate patterns explode up to 633K, causing memory overflows (>30GB) in all cases. Notably, LLMT's vanilla e-graph AU process still runs out of memory even if we adopt RII's phase-oriented equality saturation for it, due to the essentially intractable complexity. With RII features, all benchmarks completed within 145 seconds, using no more than 799MB of memory. The comparison demonstrates that RII techniques qualify e-graph AU for tractable instruction identification.

7.1.2 Comparison with baselines.

We compare ISAMORE with three baseline approaches: ENUM, which implements fine-grained convex subgraph enumeration inspired by [22, 29], NOVIA [59], which produces coarse-grained accelerators through syntactic merging, and NoEqSat, which skips EqSat in RII `Default` mode to disable semantic consideration. For

Table 3. Statistics for NOVIA and ISAMORE modes.

	Count	Size	Reuse	Runtime	Memory
NOVIA	1.8	23.2	3.2	0.4s	4MB
AstSize	5.2	9.4	4.7	116.1s	167MB
Default	7.8	8.0	6.6	19.9s	116MB
KDSample	7.8	8.4	6.4	109.3s	696MB
Vector	11	12.5	3.5	34.0s	124MB

ISAMORE, we run the Default mode with vectorization disabled. We update the latest NOVIA version (1.5.0, 2024) with ISAMORE's profiling-driven cost model (Section 5.4.1) for fair comparison. We configure the approaches to uniformly adopt NOVIA's loose I/O constraints with RoCC [7]-style memory system access support. We add an *All* benchmark that composes all nine kernels, individually repeated for a close time portion, to evaluate a multi-kernel scenario.

Figure 10 shows the evaluation results. ISAMORE consistently achieves higher speedups than baselines with moderate area overheads. The NOVIA solutions cannot achieve ideal speedup in most benchmarks, even with much larger area, especially for *QRDecomp* and *All*, because NOVIA offloads whole basic blocks, which requires a large area and contains instruction sequences that run faster on the processor of a higher clock frequency, worsening performance. NOVIA accelerators also have low reusability as shown in Table 3. ISAMORE's max-speedup solutions achieve 1.52× average speedup over NOVIA's, with speedups ranging from 1.12× to 1.94×. Both ENUM and NoEqSat can identify fine-grained patterns. Across all the benchmarks, ENUM requires more area to achieve a similar speedup as ISAMORE, since ENUM generates duplicated instructions of rare differences, demonstrating the importance of reusability-guided identification. Besides, ISAMORE achieves higher performance gains even with less area than NoEqSat. For instance, ISAMORE achieves 2.02× for *Deriche*, higher than NoEqSat's 1.51×, while saving 46.7% area. Across the benchmarks, ISAMORE's maximum speedup is on average 1.12× higher than NoEqSat's, whereas the average area is 84.9% of NoEqSat's. This trend demonstrates that ISAMORE exploits semantic equivalence for more acceleration.

7.1.3 Comparison of different modes. We evaluate more modes of ISAMORE to understand the impacts of the RII features. Derived from Default, AstSize mode uses the hardware-agnostic term size as the selection and extraction objective, KDSample mode uses the *kd-tree* pattern sampling strategy, and Vector mode runs the *pattern vectorization* in the first phase. Figure 11 shows their achieved maximum speedups across the benchmarks, and Table 3 shows the solutions' statistics, including custom instruction count, number of operations per instruction (size), reuse factor per instruction, etc. AstSize mode achieves the worst performance gains for all benchmarks, demonstrating the significance of hardware-aware selection and extraction. KDSample mode outperforms Default on *QProd*, *Deriche*, and *All*, because it samples more representative patterns and potentially identifies larger patterns that accelerate more instructions per trigger. KDSample mode's disadvantage is the long exploration time and high memory usage, as shown in Table 3.

To evaluate the *pattern vectorization* features, we further enable vectorized access to the memory system from a custom instruction unit, as the Hwacha [40] vector unit does. Vector mode outperforms Default on 8 out of 10 benchmarks, with notable improvements on *MatMul*, *MatChain*, and *QRDecomp*. One exception is *2DConv*, whose DLP potentials are not exploited. The reason is that LLVM does not apply *if-conversion* [5] to expose operations surrounded by bounds checking, hindering vectorization. Although the improvements over Default are moderate (up to 1.68×), because the loose I/O constraint allows DLP operations to also appear in wide scalar custom instructions, ISAMORE's performance superiority over NOVIA and ENUM rises to average 1.76× (up to 2.69×) and 1.46× (up to 1.95×) with vectorization enabled, demonstrating *pattern vectorization*'s effectiveness.

In addition, ISAMORE's structured DSL encodes control flow constructs in e-graph, which enables the identification of reusable hardware loops. We further evaluate the *MatChain* benchmark with the matrix multiply function inlined twice, leading to two common loop constructs. ISAMORE identifies and selects the matrix multiply's innermost loop, which LLVM partially unrolls, as a common pattern. It generates an accelerator of loop pipeline architecture with memory accesses inside the loop body vectorized, achieving 50.52× speedup compared to general processor execution. Identifying reusable hardware loops is beyond the scope of the baseline approaches, demonstrating ISAMORE's general identification granularity.

7.2 Real-World Case Studies

We evaluate ISAMORE by multiple case studies, including analyzing three open-source libraries across different domains and specializing processors for quantized LLM inference and post-quantum cryptography workloads.

7.2.1 Domain-specific libraries. This section studies the following domains: digital signal processing, image processing, and point cloud processing. We pick open-source C/C++ libraries with rich examples as the representative applications to run ISAMORE. The *liquid-dsp* provides DSP primitives for software-defined radio applications on embedded platforms. We pick six representative modules, as summarized in Table 4. For profiling, we configure the GEM5 simulation using the Minor CPU model to mimic an embedded platform. We modify the CMake configuration to enable instrumentation and run GEM5 for 174 examples to profile the modules. We apply ISAMORE to each module individually. *CImg* is a self-contained C++ template image processing library. We run 30 examples with interactive visualization, and substitute

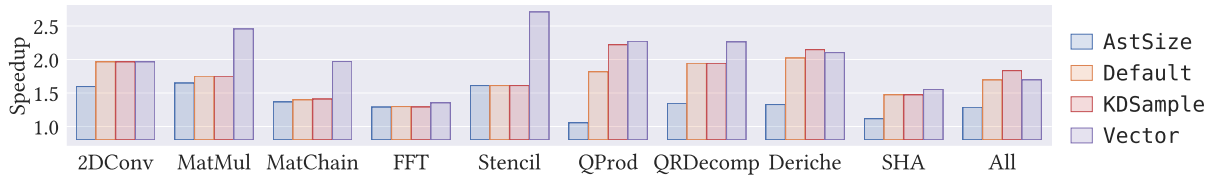
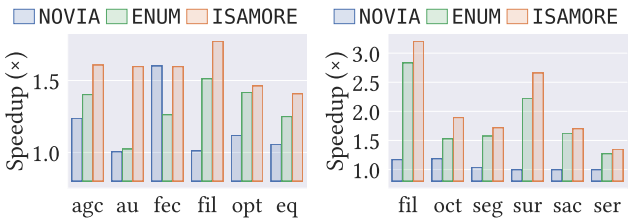


Figure 11. Maximum speedup achieved by different ISAMORE modes on the benchmarks.

Table 4. Selected *liquid-dsp* and *PCL* modules.

Module	Description	Size
agc	Automatic gain control.	1K
audio	CVSD audio encoder.	1K
fec	Forward error correction with convolutional codes, Reed-Solomon codes, etc.	5K
filter	Digital filtering capabilities with FIR, IIR, etc.	9K
optim	Gradient search and quasi-Newton methods.	2K
equalization	Adaptive equalizers: LMS, RLS, etc.	3K
filters	Filtering mechanisms including noise removal, outlier rejection, and downsampling.	9K
octree	Hierarchical spatial data structure for search, voxelization, and neighborhood queries.	9K
segment	Segmenting point clouds into clusters.	3K
surface	Reconstructing the original surfaces.	5K
sac	Random Sample Consensus (RANSAC).	6K
search	Searching for nearest neighbors in point clouds.	7K



(a) *liquid-dsp* results **(b)** *PCL* results

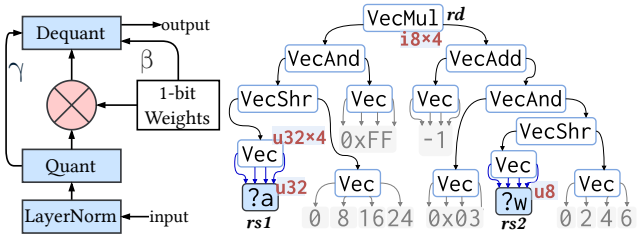
	Speedup	Area	Count	Size	Reuse
NOVIA	1.01×	10314μm^2	1	167	8
ISAMORE	1.18×	975μm^2	8	3	93

(c) *CImg* results.

Figure 12. Experimental results for real-world libraries.

the slow GEM5 simulation with a fast LLVM-based profiling execution. The original e-graph for analysis contains 45K e-classes. *PCL* is a large-scale project for point cloud processing. We adopt the same profiling methodology as *CImg*, and apply ISAMORE to six modules individually, with the original e-class sizes up to 9K, as shown in Table 4.

We run ISAMORE in Vector mode for the libraries. For *liquid-dsp*, Figure 12a shows that ISAMORE achieves higher speedup than NOVIA on the modules except *fec*, for which NOVIA merges the computation of different FEC codes, achieving ideal acceleration results. On average, ISAMORE achieves



(a) *BitLinear* **(b)** Identified pattern to become CI

Figure 13. Case study on BitNet.

1.39× speedup over NOVIA with 84.0% area saving and outperforms ENUM by 1.21×. On the monolithic *CImg* library, Figure 12c shows the comparison, where NOVIA generates one huge custom hardware unit of size 167 from merging eight basic blocks, and ISAMORE identifies eight custom instructions, with an average reuse count of 93. Detailed analysis shows that ISAMORE-identified instructions perform pixel modification based on computed indices, conditional operations with mask generation, and type conversions. Such highly reusable instructions are valuable for ISA specialization. ISAMORE's identified instructions achieve 1.18× speedup with only 975 μm^2 area, saving 90.5% area than NOVIA, which achieves merely 1.01× speedup. For the *PCL* modules, ISAMORE achieves 1.64× average speedup over NOVIA (up to 2.73×) with 93.2% area saving and outperforms ENUM by 1.18×. Overall, ISAMORE consistently delivers better performance across applications.

7.2.2 Quantized LLM.. We further conduct a comprehensive case study on the 1-bit Language Language Model (LLM) BitNet b1.58 [44]'s inference, to evaluate ISAMORE's effectiveness for processor specialization. It adopts a ternary weight representation $\{-1, 0, +1\}$ and uses *BitLinear*, as illustrated in Figure 13a, to replace conventional matrix multiplication in the Transformer [61] architecture. While this design offers significant potential for performance gains, its real-world benefits are often limited by the lack of specialized hardware units for low-bit operations. For example, dot product between 8-bit inputs and 2-bit weights on a general processor either uses inefficient multiply-then-add (MAD) instructions or needs a carefully crafted software look-up table.

We analyze a MAD-based implementation of a *BitLinear* originated from `bitnet.cpp` [65] using ISAMORE and reveals a vectorized pattern computing packed low-bit dot product,

as depicted in Figure 13b, through RII's e-graph AU flow and pattern vectorization. ISAMORE's backend generates a hardware description for the custom unit with the RoCC [7] wrapper. It configures a Rocket tile [6, 7] to instantiate the custom unit and runs Verilator [64] for RTL simulation to get precise performance reports, showing a 2.15× speedup for *BitLinear* over the baseline Rocket tile. The speedup is moderate due to the IO bandwidth constraint (32 bits per scalar register) on the vectorization length. ISAMORE also runs the OpenROAD flow for physical design, reporting 4.81% area overhead and no frequency decrease at 161.29MHz.

7.2.3 Post-Quantum Cryptography. Extending ISA for PQC is recognized as an important task by the RISC-V security standing committee [36] and has been explored by prior works [26, 27, 38]. As the standardization process of PQC is still ongoing, it's important to automate hardware acceleration for newly evolving algorithms. We study CRYSTALS-KYBER [11], a standardized key encapsulation mechanism (KEM), defined over the polynomial ring $\mathbb{Z}_q[x]/\langle x^N + 1\rangle$ with the polynomial multiplication as the computational bottleneck. We run ISAMORE for a CRYSTALS-KYBER implementation with Number Theoretic Transform (NTT) to reduce polynomial multiplication's complexity, under the same configuration as Section 7.2.2. ISAMORE identifies a custom instruction corresponding to the *butterfly* operation, which is reused by forward NTT and inverse NTT. It implements a RoCC accelerator for the Rocket tile system, achieving 5.15× speedup according to RTL simulation. OpenROAD reports 17.67% area overhead due to expensive hardware multipliers, and 2.58% frequency decrease. This case study demonstrates ISAMORE's potential for automated hardware specialization to keep pace with software evolution.

8 Related Work

Algorithms for enumerating dataflow subgraphs for fine-grained custom instructions [8, 17, 21, 22, 29, 52, 53, 55] have been well studied to handle convexity and I/O constraints. [3, 10] explored instruction recurrence; they adopt graph isomorphism to filter the enumerated subgraphs rather than guide enumeration with reusability as RII does. [3, 32] introduces canonicalized representations to unveil more syntactic merging opportunities. RII is the first to consider semantic equivalence. C-Cores and GreenDroid [35, 62] offload functions and loops to custom accelerators, and QsCores [63] mines syntactic common patterns. They exclusively support coarse-grained specialization, while ISAMORE supports more general granularity. PICO [2] and FINDER [30] identify parallel custom instructions for VLIW architectures, while APEX [46] and RADISH [66] customize processing elements from common patterns for domain-specific reconfigurable accelerators, both of which are potential stages for ISAMORE.

E-graph is a powerful data structure for rewrite-based optimization. Diospyros [60] vectorizes DSP kernels to exploit existing parallel instructions via equality saturation, and Isaria [58] extends it with automatic rule synthesis to automate the construction of rewrite-based compilers. ISAMORE solves a complementary problem, identifying new custom instructions for an application domain. It is an open problem to compose Isaria and ISAMORE for both instruction customization and compiler generation in an exploration loop. E-graph techniques are also used for high-level synthesis [19] and logic synthesis [14, 15, 24, 71], which can be absorbed into ISAMORE for better hardware implementation.

9 Conclusion

This paper presents ISAMORE, an end-to-end framework that implements the RII methodology for reusable instruction customization. It introduces scalable e-graph anti-unification to identify semantic-aware common patterns and exploit data-level parallelism for vectorized custom instructions. ISAMORE outperforms baseline approaches by up to 2.69× on benchmarks, and its practical effectiveness is demonstrated through rich case studies.

Acknowledgments

We would like to thank our anonymous reviewers for their constructive feedback. We especially thanks Shoaib Kamil for shepherding our paper.

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

References

[1] 2025. eembc/coremark-pro. https://github.com/eembc/coremark-pro original-date: 2019-07-19T18:34:34Z.

[2] S. Aditya, B. Ramakrishna Rau, and V. Kathail. 1999. Automatic architectural synthesis of VLIW and EPIC processors. In *Proceedings 12th International Symposium on System Synthesis*. 107–113. https://doi.org/10.1109/ISSS.1999.814268

[3] Junwhan Ahn and Kiyoung Choi. 2013. Isomorphism-Aware Identification of Custom Instructions With I/O Serialization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 1 (Jan. 2013), 34–46. https://doi.org/10.1109/TCAD.2012.2214033

[4] Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, Mingyu Woo, and Bangqi Xu. 2019. Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/3316781.3326334

[5] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '83)*. Association for Computing Machinery, New York, NY, USA, 177–189. https://doi.org/10.1145/567067.567085

[6] 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* (2020). https://doi.org/10.1109/MM.2020.2996616

[7] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, and others. 2016. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016). https://aspire.eecs.berkeley.edu/wp/wp-content/uploads/2016/04/Tech-Report-The-Rocket-Chip-Generator-Beamer.pdf

[8] Kubilay Atasu, Wayne Luk, Oskar Mencer, Can Ozturan, and Günhan Dundar. 2012. FISH: Fast Instruction SyntHesis for Custom Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20, 1 (Jan. 2012), 52–65. https://doi.org/10.1109/TVLSI.2010.2090543

[9] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517. https://doi.org/10.1145/361002.361007

[10] Paolo Bonzini and Laura Pozzi. 2008. Recurrence-Aware Instruction Set Selection for Extensible Embedded Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16, 10 (Oct. 2008), 1259–1267. https://doi.org/10.1109/TVLSI.2008.2001863

[11] Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. 2018. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 353–367. https://doi.org/10.1109/EuroSP.2018.00032

[12] Cadence Design Systems, Inc. 2025. Cadence Tensilica Offerings. https://www.cadence.com/en_US/home/tools/silicon-solutions/compute-ip/technologies.html Publication Title: Cadence Tensilica Offerings.

[13] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2022. babble: Learning Better Abstractions with E-Graphs and Anti-Unification. https://doi.org/10.48550/arXiv.2212.04596 arXiv:2212.04596.

[14] Chen Chen, Guangyu HU, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. 2025. E-morphic: Scalable Equality Saturation for Structural Exploration in Logic Synthesis. https://doi.org/10.48550/arXiv.2504.11574 arXiv:2504.11574 [cs].

[15] Chen Chen, Guangyu Hu, Dongsheng Zuo, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. 2024. E-Syn: E-Graph Rewriting with Technology-Aware Cost Functions for Logic Synthesis. https://doi.org/10.48550/arXiv.2403.14242 arXiv:2403.14242 [cs].

[16] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Allo: A Programming Model for Composable Accelerator Design* 8, PLDI (June 2024), 171:593–171:620. https://doi.org/10.1145/3656401

[17] Xiaoyong Chen, Douglas L. Maskell, and Yang Sun. 2007. Fast Identification of Custom Instructions for Extensible Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (Feb. 2007), 359–368. https://doi.org/10.1109/TCAD.2006.883915

[18] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (Jan. 2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[19] Jianyi Cheng, Samuel Coward, Lorenzo Chelini, Rafael Barbalho, and Theo Drane. 2024. SEER: Super-Optimization Explorer for High-Level Synthesis using E-graph Rewriting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24, Vol. 2)*. Association for Computing Machinery, New York, NY, USA, 1029–1044. https://doi.org/10.1145/3620665.3640392

[20] CIRCT community. 2025. CIRCT. https://circt.llvm.org/

[21] N. Clark, Hongtao Zhong, and S. Mahlke. 2003. Processor acceleration through automated instruction set customization. In *22nd Digital Avionics Systems Conference. Proceedings (Cat. No.03CH37449)*. IEEE Comput. Soc, San Diego, CA, USA, 129–140. https://doi.org/10.1109/MICRO.2003.1253189

[22] N.T. Clark, H. Zhong, and S.A. Mahlke. 2005. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Trans. Comput.* 54, 10 (Oct. 2005), 1258–1270. https://doi.org/10.1109/TC.2005.156 Conference Name: IEEE Transactions on Computers.

[23] Codasip. 2025. Codasip Studio. https://codasip.com/products/codasip-studio/ Publication Title: Codasip.

[24] Samuel Coward, Theo Drane, Emiliano Morini, and George Constantinides. 2024. Combining Power and Arithmetic Optimization via Datapath Rewriting. https://doi.org/10.48550/arXiv.2404.12336 arXiv:2404.12336 [cs].

[25] David Tschumperlé and others. 2025. The CImg Library, a small and open-source C++ toolkit for image processing. https://github.com/GreycLab/CImg

[26] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. 2022. Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2022), 414–460. https://doi.org/10.46586/tches.v2022.i1.414-460

[27] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. 2020. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Aug. 2020), 239–280. https://doi.org/10.13154/tches.v2020.i4.239-280

[28] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. 2017. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (Oct. 2017). https://doi.org/10.1109/TVLSI.2017.2654506

[29] Emanuele Giaquinta, Anadi Mishra, and Laura Pozzi. 2015. Maximum Convex Subgraphs Under I/O Constraint for Automatic Identification of Custom Instructions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 3 (March 2015), 483–494. https://doi.org/10.1109/TCAD.2014.2387375

[30] Vikkitharan Gnanasambandapillai, Jorgen Peddersen, Roshan Ragel, and Sri Parameswaran. 2020. FINDER: Find Efficient Parallel Instructions for ASIPs to Improve Performance of Large Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (Nov. 2020), 3577–3588. https://doi.org/10.1109/TCAD.2020.3012211 Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[31] Cecilia González-Álvarez, Jennifer B. Sartor, Carlos Álvarez, Daniel Jiménez-González, and Lieven Eeckhout. 2013. Accelerating an application domain with specialized functional units. *ACM Trans. Archit. Code Optim.* 10, 4 (2013), 47:1–47:25. https://doi.org/10.1145/2541228.2555303

[32] Cecilia González-álvarez, Jennifer B. Sartor, Carlos Álvarez, Daniel Jiménez-González, and Lieven Eeckhout. 2016. MInGLE: An Efficient Framework for Domain Acceleration Using Low-Power Specialized Functional Units. *ACM Trans. Archit. Code Optim.* 13, 2 (2016), 17:1–17:26. https://doi.org/10.1145/2898356

[33] Google Inc. 2025. XLS: Accelerated HW Synthesis. https://google.github.io/xls/

[34] Gordon Plotkin. 1970. *Lattice Theoretic Properties of Subsumption.* Edinburgh University, Department of Machine Intelligence and Perception.

[35] Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha

Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, and Michael Taylor. 2011. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro* 31, 2 (March 2011), 86–95. https://doi.org/10.1109/MM.2011.18

[36] Helena Handschuh. 2020. RISC-V: An Open Approach to System Security – RISC-V International. https://riscv.org/blog/2020/03/risc-v-an-open-approach-to-system-security/

[37] John C. Reynolds. 1970. Transformational systems and the algebraic structure of atomic formulas. *Edinburgh University Press* 5 (1970), 135–152.

[38] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. 2024. Post-Quantum Signatures on RISC-V with Hardware Acceleration. *ACM Trans. Embed. Comput. Syst.* 23, 2 (2024), 30:1–30:23. https://doi.org/10.1145/3579092

[39] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '04)*. IEEE Computer Society, USA, 75.

[40] Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and Krste Asanović. 2015. The Hwacha Microarchitecture Manual, Version 3.8. (2015). https://people.eecs.berkeley.edu/~krste/papers/EECS-2015-263.pdf

[41] LLVM Community. 2025. LLVM 18.1.8. https://github.com/llvm/llvm-project/releases

[42] Louis-Noel Pouchet and Tomofumi Yuki. 2018. PolyBench/C 4.2. https://sourceforge.net/projects/polybench/

[43] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. https://doi.org/10.48550/arXiv.2007.03152 arXiv:2007.03152 [cs].

[44] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024. The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits. https://doi.org/10.48550/arXiv.2402.17764 arXiv:2402.17764 [cs].

[45] 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 (Jan. 2021), 1–29. https://doi.org/10.1145/3434304

[46] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Ritvik Sharma, Clark Barrett, Mark A. Horowitz, Pat Hanrahan, and Priyanka Raina. 2023. APEX: A Framework for Automated Processing Element Design Space Exploration using Frequent Subgraph Analysis. 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, New York, NY, USA, 33–45. https://doi.org/10.1145/3582016.3582070

[47] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2025. Large Language Models: A Survey. https://doi.org/10.48550/arXiv.2402.06196 arXiv:2402.06196 [cs].

[48] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3385412.3385974

[49] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3445814.3446712

[50] 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. https://doi.org/10.1145/3620666.3651375

[51] Anjali Pal, Brett Saiki, Ryan Tjoa, Cynthia Richey, Amy Zhu, Oliver Flatt, Max Willsey, Zachary Tatlock, and Chandrakana Nandi. 2023. Equality Saturation Theory Exploration à la Carte. *Reproduction package for "Equality Saturation Theory Exploration à la Carte"* 7, OOPSLA2 (2023), 258:1034–258:1062. https://doi.org/10.1145/3622834

[52] Pan Yu, Pan Yu, Tulika Mitra, and Tulika Mitra. 2004. Scalable custom instructions identification for instruction-set extensible processors. (Sept. 2004), 69–78. https://doi.org/10.1145/1023833.1023844 MAG ID: 2121156724.

[53] L. Pozzi, K. Atasu, and P. Ienne. 2006. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 7 (July 2006), 1209–1229. https://doi.org/10.1109/TCAD.2005.855950 Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[54] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 110–119. https://doi.org/10.1109/IISWC.2014.6983050

[55] J. Reddington, G. Gutin, A. Johnstone, E. Scott, and A. Yeo. 2009. Better Than Optimal: Fast Identification of Custom Instruction Candidates. In *2009 International Conference on Computational Science and Engineering*, Vol. 2. 17–24. https://doi.org/10.1109/CSE.2009.167

[56] Nico Reissmann, Jan Christian Meyer, Helge Bahmann, and Magnus Själander. 2020. RVSDG: An Intermediate Representation for Optimizing Compilers. http://arxiv.org/abs/1912.05036 arXiv:1912.05036.

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

[58] Samuel Thomas and James Bornholt. 2024. Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24, Vol. 1)*. Association for Computing Machinery, New York, NY, USA, 19–34. https://doi.org/10.1145/3617232.3624873

[59] 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. ACM, Virtual Event Greece, 507–521. https://doi.org/10.1145/3466752.3480094

[60] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. https://doi.org/10.48550/arXiv.1706.03762 arXiv:1706.03762 [cs].

[62] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation cores: reducing the energy of mature computations. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 205–218. https://doi.org/10.1145/1735970.1736044

[63] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. 2011. QsCores: trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/2155620.2155640

[64] Veripool. 2025. Verilator. https://www.veripool.org/verilator/

[65] Jinheng Wang, Hansong Zhou, Ting Song, Shijie Cao, Yan Xia, Ting Cao, Jianyu Wei, Shuming Ma, Hongyu Wang, and Furu Wei. 2025. Bitnet.cpp: Efficient Edge Inference for Ternary LLMs. https://doi.org/10.48550/arXiv.2502.11880 arXiv:2502.11880 [cs].

[66] Max Willsey, Vincent T. Lee, Alvin Cheung, Rastislav Bodík, and Luis Ceze. 2019. Iterative Search for Reconfigurable Accelerator Blocks With a Compiler in the Loop. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 3 (March 2019), 407–418. https://doi.org/10.1109/TCAD.2018.2878194

[67] Youwei Xiao, Fan Cui, Zizhang Luo, Weijie Peng, and Yun Liang. 2025. Cayman: Custom Accelerator Generation with Control Flow and Data Access Optimization. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. 1–7. https://doi.org/10.1109/DAC63849.2025.11132875

[68] 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. https://doi.org/10.1145/3626202.3637561

[69] 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)*. 1–9. https://doi.org/10.1109/ICCAD66269.2025.11240817 ISSN: 1558-2434.

[70] 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*. ACM, San Diego California, 1–9. https://doi.org/10.1145/3508352.3549370

[71] Jiaqi Yin, Zhan Song, Chen Chen, Qihao Hu, and Cunxi Yu. 2025. BoolE: Exact Symbolic Reasoning via Boolean Equality Saturation. https://doi.org/10.48550/arXiv.2504.05577 arXiv:2504.05577 [cs].

[72] Hengrui Zhang, August Ning, Rohan Baskar Prabhakar, and David Wentzlaff. 2025. LLMCompass: Enabling Efficient Hardware Design for Large Language Model Inference. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*. IEEE Press, Buenos Aires, Argentina, 1080–1096. https://doi.org/10.1109/ISCA59077.2024.00082

[73] Yuyang Zou, Youwei Xiao, Yansong Xu, Chenyun Yin, Yuhao Luo, Yitian Sun, Ruifan Xu, Renze Chen, and Yun Liang. 2025. Aquas: Enhancing Domain Specialization through Holistic Hardware-Software Co-Optimization based on MLIR. https://doi.org/10.48550/arXiv.2511.22267 arXiv:2511.22267 [cs].