NUKE: An Automatic Pattern Fusion Framework for CGRAs using E-graphs

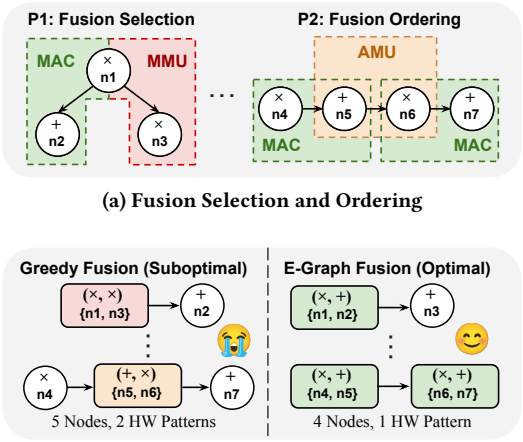
Jiajun Qin[1], Yuhao Ji[1], Zhaotan Lin[1], Cheng Tan[2,3], Xinyun Zhang[4], Bei Yu[1]
[1]The Chinese University of Hong Kong [2]Google
[3]Arizona State University [4]ShanghaiTech University

Abstract

Pattern fusion is a key optimization for coarse-grained reconfigurable architectures, as it can reduce graph size, relieve resource pressure, and shorten recurrence-critical paths during mapping. However, existing fusion approaches either rely on manually designed patterns or use greedy heuristics, making them vulnerable to the phase-ordering problem and often insensitive to hardware cost. To address this, we propose NUKE, an automatic pattern fusion framework based on equality graphs. NUKE combines algebraic rewrites with dynamically mined hardware-aware fusion rules, and uses a two-stage extraction procedure that first fixes cycle-critical structures and then reduces hardware cost for the remaining off-cycle operations. Our evaluations demonstrate that NUKE not only improves performance but also provides favorable area-efficiency. Our framework achieves a 1.46× average speedup over the unfused baseline with 1.16× area-efficiency, while closely approaching manual fusion.

1 Introduction

Coarse-grained reconfigurable architectures (CGRAs), which typically consist of an M×N array of tiles composed of programmable functional units (FUs), have demonstrated significant potential as domain-specific accelerators across various high-impact application domains [1–6]. However, achieving efficient deployment remains challenging because the vast design space involved in customizing the architecture for diverse applications necessitates a careful balance between flexibility and efficiency. A critical step in the exploration involves determining the optimal tile microarchitecture, which must be tailored to meet the specific requirements of target applications while ensuring that hardware benefits are not offset by excessive resource overhead.



(a) Fusion Selection and Ordering

(b) Greedy Approach v.s. Our Approach

Figure 1: Illustration of the phase-ordering problem in pattern fusion.

Pattern fusion serves as a pivotal technique in this design process because it significantly improves mapping quality and execution efficiency by consolidating frequently occurring subgraphs into dedicated hardware operators [7–12]. However, the efficacy of this technique relies on the non-trivial task of identifying an optimal set of patterns that maximizes performance while strictly constraining hardware costs such as area and power, which requires a holistic exploration framework for balancing the trade-offs between architectural complexity and performance gains.

Existing approaches to pattern fusion generally fall into manual or automated categories, where manual extraction relies on designers identifying frequent patterns based on domain knowledge [7, 8, 12–14], which is inherently labor-intensive and difficult to scale to modern application suites. To address these scalability constraints, automated heuristic-based methods have been developed to extract patterns directly from target workloads [9–11, 15]. However, their reliance on greedy algorithms without global context often leads to suboptimal fusion decisions.

A primary challenge introduced by these greedy heuristics is the phase-ordering problem, which is illustrated in Figure 1

This work is supported by The Research Grants Council of Hong Kong SAR (No. CUHK14211824, CUHK14210723, and T46-415/25-R).

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICCAD '26, San Jose, CA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/10.1145/3831252.3833947

and describes how early fusion decisions can irreversibly preclude the formation of more beneficial patterns later in the process. Because greedy algorithms commit to local optima without evaluating the entire design space, they often fail to discover globally optimal patterns that would provide superior performance gains across the entire DFG. Another critical challenge is that purely graph-based heuristics often lack sufficient hardware-awareness during the extraction phase, which frequently leads to the generation of overly complex patterns that result in excessive hardware overhead.

To address these challenges, we propose **NUKE**, which is an automatic patterN fUsion frameworK for CGRAs that utilizes E-graphs. Rather than making irreversible fusion decisions, NUKE compactly represents equivalent fusion alternatives in a shared equality-saturation structure and extracts the solution that best balances loop latency with hardware overhead through a specialized cost model.

Our key contributions are summarized as follows:

- We propose NUKE, the first E-graph based framework for automatic pattern fusion in CGRAs, enabling global exploration of interacting fusion opportunities without the phase-ordering limitations of greedy heuristics.
- We develop a hardware-aware fusion flow that combines hybrid rewrite-rule generation with a cycle-aware two-stage extractor to reduce recurrence path while maintaining efficient hardware units.
- We demonstrate through comprehensive evaluation that NUKE outperforms baselines across both kernels and applications, scales effectively to larger fabrics, and achieves performance comparable to manual designs.

2 Background and Motivation

2.1 CGRA and its Mapping

CGRAs are programmable accelerators consisting of an array of tiles, integrated with host CPUs in either a loosely coupled [16–18] or a tightly coupled [19, 20] manner. Our design adopts the loosely coupled approach, where the CGRA independently operates while the CPU handles data movement and instruction dispatch.

As shown in Figure 2(a), each tile in the array contains multiple FUs for computation, register files for temporal data storage, configuration memory, and on-chip interconnects for inter-tile communication. The compiler generates per-cycle configuration signals that direct each tile to execute the corresponding operations and control data routing.

To process a given application kernel, the compiler first derives its DFG and then employs modulo scheduling [21, 22] to assign the nodes onto the CGRA tiles. The II, determined by the compiler, denotes the number of cycles between successive loop iteration launches. For loops with a large iteration count, the II dominates overall execution latency, making it the primary optimization target for CGRA compilation.

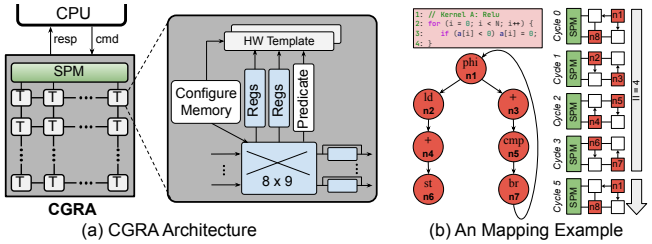


Figure 2: Overview of the CGRA architecture and its mapping process. (a) The spatial hardware architecture and tile internal components. (b) An example of mapping a relu kernel onto the CGRA.

2.2 Pattern Fusion

Pattern fusion is a widely adopted technique in CGRA design where frequently occurring subgraphs within the DFG are identified as computational patterns and merged into single coarse-grained nodes, which are subsequently executed by dedicated hardware FUs. By consolidating multiple primitive operations into a unified pattern, this fusion process significantly reduces the total DFG node count to lower the resource demand (ResMII) while potentially shortening critical recurrence paths to reduce the recurrence constraint (RecMII). However, it also introduces a fundamental tradeoff between performance improvement and area overhead since each specialized pattern requires additional hardware resources within the processing tile.

A central difficulty in automating pattern fusion is the **phase-ordering** problem, which manifests in two aspects. (1) *Fusion selection*: Given that a single DFG node may participate in multiple candidate patterns, fusing this node into one pattern inherently restricts the fusion opportunities for its adjacent nodes. As illustrated in the P1 scenario of Figure 2(a), node n1 can be grouped with node n2 to form a MAC unit or with node n3 to form an MMU, which means selecting one mapping discards the alternative. (2) *Fusion ordering*: candidate patterns may overlap in the DFG, where committing to one fusion early can preclude another pattern from being applied. The P2 scenario in Figure 2(a) shows how different fusion candidates compete for a shared chain of nodes, meaning that the fusion ordering determines the final DFG quality. Figure 2(b) further illustrates the consequences of failing to resolve these challenges, where a sequential greedy approach traps the compiler in a local minimum yielding five nodes. In contrast, an optimal exploration strategy resolves both selection and ordering conflicts to discover a four-node configuration that minimizes execution latency and hardware utlization.

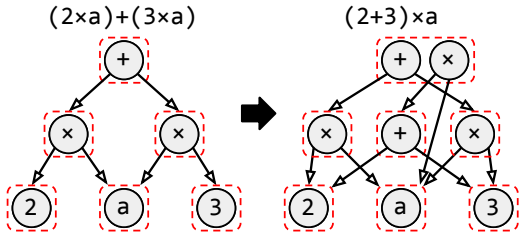


Figure 3: An E-graph produced by applying standard arithmetic rewrites. Starting from $(x \times 2) + (x \times 3)$, distributivity exposes the simpler factored form $x \times (2 + 3)$, and constant folding reduces it to $x \times 5$. Each gray node is an e-node, and each red dashed box is an e-class.

Beyond the phase-ordering problem, the efficient evaluation of candidate patterns presents another practical challenge. Exhaustively mapping and simulating every possible fused DFG variant to obtain exact hardware metrics is computationally prohibitive. An effective fusion framework must therefore incorporate lightweight cost estimation mechanisms to guide pattern selection without requiring full compilation passes. NUKE addresses both of these challenges within a unified equality-graph framework. This approach simultaneously explores equivalent fusion structures and evaluates their associated hardware costs.

2.3 E-Graph

An E-graph (equality graph) is a compact data structure that simultaneously represents a large set of equivalent expressions [23–25]. It consists of two core components: *e-classes* and *e-nodes*. An e-node encodes a specific operator applied to its operands, while an e-class groups all e-nodes that are semantically equivalent. This hierarchical structure allows the E-graph to share common subexpressions across the entire equivalence space without materializing each variant individually. Figure 3 shows this idea on a slightly richer arithmetic example, where distributivity exposes a common factor and constant folding further simplifies the expression.

The principal advantage of the E-graph lies in its ability to circumvent the phase-ordering problem. Conventional compilers apply rewrite rules sequentially and destructively, where each transformation commits to a single choice and may preclude beneficial rewrites downstream. In contrast, the E-graph applies rewrite rules additively through a process called *equality saturation*: each rule application introduces new equivalent expressions into the graph without discarding existing ones. Once saturated, an extraction algorithm selects the globally optimal expression from the full equivalence space under a given cost model, effectively evaluating all possible rewrite orderings simultaneously rather than committing to a fixed sequence.

These compelling advantages directly inspire our approach to resolving the bottlenecks in pattern fusion. We recognize that formulating hardware patterns as rewrite rules allows an E-graph to systematically explore and maintain all possible fused DFG configurations. Motivated by this insight, we propose the NUKE framework, which utilizes E-graphs to perform automated pattern fusion. During the final extraction phase, NUKE evaluates these maintained configurations by estimating their impact on the critical path alongside hardware overhead to finalize globally optimal fusion decisions.

3 Methodology

In this section we introduce our NUKE design. Section 3.1 provides an overview of the complete framework pipeline. Section 3.2 details the core E-graph fusion engine, including bidirectional DFG/S-expression conversion, hybrid rule generation, and cycle-aware extraction. Finally, Section 3.3 describes the DFG mapping procedure and parameterized FU library for end-to-end CGRA evaluation.

3.1 Overview

Figure 4 illustrates the end-to-end workflow of NUKE. Given application kernels as input, the framework proceeds through the following stages:

❶ **Loop Unrolling**. The input kernel is first subject to loop unrolling with a configurable unrolling factor (UF) to expose instruction-level parallelism across loop iterations, enlarging the DFG and creating more opportunities for pattern fusion.

❷ **DFG Generation**. The unrolled kernel is then lowered into a DFG representation built on top of the MLIR infrastructure, where each node corresponds to a primitive operation and edges encode data dependencies.

❸ **E-graph based Fusion**. This is the core optimization stage of NUKE. The generated DFG is first converted into S-expressions and, together with automatically constructed rewrite rules, fed into the E-graph engine for equality saturation. A cycle-aware extraction algorithm then selects the optimal variant from the saturated E-graph, which is reconstructed back into a fused DFG. The details of this stage are presented in Section 3.2.

❹ **DFG Mapping**. The fused DFG is mapped onto the target CGRA using the modular scheduling [21]. To support the diverse fused patterns produced by the E-graph, we develop a parameterized template FU library that dynamically instantiates specialized hardware FUs for each extracted pattern, ensuring architectural compatibility without manual hardware design effort. Details can be found in Section 3.3.

❺ **Evaluation**. The compiled results, including II, cycle count, and measured PPA, are collected to evaluate the quality of the fused DFG. This stage serves both as the final output and as a feedback mechanism for tuning fusion parameters.

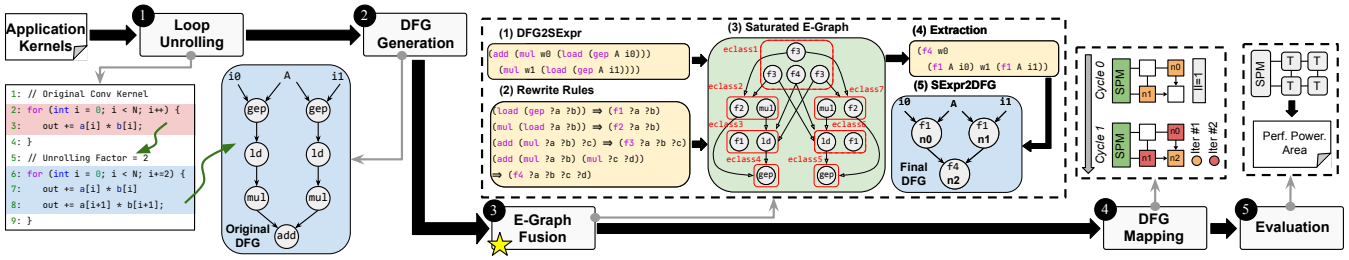


Figure 4: Overall compiler toolchain of NUKE. The workflow starts from application kernels and proceeds through loop transformation, DFG generation, E-graph based fusion, DFG mapping, and evaluation.

3.2 E-graph based Fusion

3.2.1 Conversion between DFG and S-Expression. As the E-graph engine relies on S-expressions for its underlying representation [25], we implement a bidirectional translation interface between the input DFG and this symbolic format to fully utilize its rewriting capabilities.

Forward Translation (DFGToSExpr). The forward pass serializes the DFG into prefix-notation S-expressions by traversing the use-def chains from each root node. To ensure structural uniqueness in the E-graph, we canonicalize node references: rather than using raw memory addresses, operands are mapped to normalized symbolic variables (e.g., `v0`, `v1`) based on their topological order in the DFG.

A critical challenge in serializing a graph to an expression tree is handling nodes that produce multiple results. Our approach maps the node's S-expression string to its primary result in the symbol table. If a node yields multiple values, the serializer associates the generated S-expression with the defining node instance. Subsequent references to any of the node's results resolve to the same canonicalized string, maintaining the logical link between the diverse outputs and their single source node.

Backward Reconstruction (SExprToDFG). As shown in Step (5) of Figure 4, The reverse transformation reconstructs the DFG from the optimized S-expressions using a custom recursive descent parser. This parser systematically instantiates the corresponding DFG nodes and restores their data dependencies. Crucially, when the parser encounters a `fused_op` token (e.g., `fused_op.mul_add`), it directly creates a single coarse-grained node that encapsulates the matched pattern subgraph. This step effectively replaces multiple primitive operations with one unified fused node.

3.2.2 Rewrite Rules. Rewrite rules drive design space exploration within the E-graph framework. Existing E-graph applications in logic synthesis typically rely on static and manually curated rule libraries [26]. In contrast, efficient CGRA mapping requires the identification of specialized compound operations. Relying exclusively on predefined patterns traps the compiler in local optima, preventing the system from exploiting the all possibilities of the hardware. Instead of depending on human priors, we aim to systematically traverse all possible fusion opportunities. To achieve this, our framework implements a hybrid rule generation system. This approach combines standard algebraic axioms with dynamic fusion rules derived directly from the input DFG, a pipeline we illustrate in Figure 5.

PPA-Constrained Pattern Mining. To fully exploit the potential of pattern fusion, the framework automatically extracts fusion candidates by traversing the DFG. We employ an exhaustive strategy to generate all possible patterns and filter them using performance, power, and area constraints defined by the target architecture.

As shown in the first stage of Figure 5, the mining algorithm iterates through each operation in the original DFG by treating it as the root of a potential subgraph. It generates normalized patterns by replacing concrete values with symbolic wildcards. The figure illustrates this process by extracting a multiplication and addition subgraph and normalizing its inputs into wildcards `?a`, `?b`, and `?c`. The framework then validates these candidates against the following criteria:

- Hardware Constraints: The cumulative area and critical path latency of the pattern must not exceed the physical limits of a single tile. This is calculated using a predefined hardware specification. Our cost model uses first-order analytical estimates calibrated against post-synthesis results for basic FUs (e.g., adders, multipliers, shifters) and compound FUs (e.g., MACs). For each newly mined pattern, its area is estimated by aggregating the costs of its constituent components, while its latency is estimated from the longest path within the pattern.
- Frequency Threshold: Only patterns that appear frequently across the DFG (exceeding a configurable threshold) are promoted to rewrite rules, ensuring the solver focuses on high-impact optimizations.

Dynamic Rule Synthesis. Our framework then synthesizes valid patterns into unidirectional rewrite rules. As depicted in the final stage of Figure 5, the left side (LHS) represents the graph of primitive operations likes `(add (mul ?a`

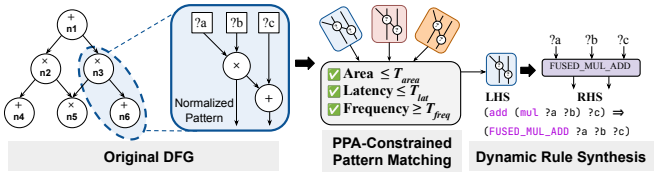


Figure 5: Rewrite rule generation pipeline. Candidate subgraphs are enumerated from the DFG, normalized with symbolic wildcards, filtered by PPA constraints, and synthesized into unidirectional rewrite rules.

`?b) ?c)`. The right side (RHS) introduces the corresponding fused operator `(FUSED_MUL_ADD ?a ?b ?c)`. Injecting these dynamically generated rules into the E-graph allows the rewriter to seamlessly explore the trade-offs between primitive instruction chains and fused coarse-grained operators within a unified mathematical space.

3.2.3 Extraction. Upon the completion of the e-graph rewriting phase, NUKE employs a specialized extraction algorithm to identify the optimal DFG from an expansive space of equivalent candidates. As illustrated in Figure 6, our extraction procedure adopts a strict two-phase sequential cost model inspired by the hierarchical optimization strategies in SEER [27]. The first phase performs cycle-aware extraction to fix the cycle-critical structure that determines the RecMII, and the second phase performs area-aware extraction to reduce the hardware cost of the remaining off-cycle operations.

Cycle-Aware Extraction. In CGRA mapping, the II strictly governs the asymptotic execution throughput of a pipelined loop. This II is fundamentally lower-bounded by the RecMII, which depends on the cumulative operation latency along the most critical recurrence cycle within the DFG. These restrictive cycles are typically induced by loop-carried data dependencies or backward control-flow edges involving `ctrl_mov` instructions. To effectively navigate the expansive space of structurally equivalent loop patterns encoded within the E-graph, our extractor systematically isolates the specific node combination on these cycle-critical subgraphs. Formally, let V be the set of nodes in a candidate DFG, and $C(v) \in \{0, 1\}$ be an indicator function that evaluates to 1 if node v is part of a cycle-critical subgraph, and 0 otherwise. The cycle-aware cost function is defined as:

$$Cost_{cycle}(G) = \sum_{v \in V} C(v). \tag{1}$$

By minimizing $Cost_{cycle}(G)$, the extractor effectively fixes the selection of cycle-critical operations to minimize RecMII.

Area-Aware Extraction. Following the determination of the optimal cycle-critical structure, the algorithm proceeds to optimize the remaining off-cycle operations. While the primary objective secures the optimal RecMII, numerous DFG candidates may exhibit the identical minimal RecMII but differ

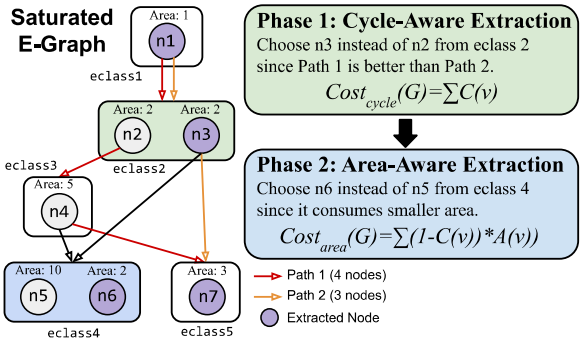


Figure 6: An example of the two-phase extraction.

significantly in their off-cycle implementations. To resolve these equivalent choices, the extractor evaluates a secondary objective focused on area minimization. Let $A(v)$ represent the hardware area cost of node v. The area-aware cost function is applied exclusively to nodes outside the predetermined cycle-critical subgraphs:

$$Cost_{area}(G) = \sum_{v \in V} (1 - C(v)) \cdot A(v). \tag{2}$$

Figure 6 provides a concrete illustration of this two-phase methodology. The saturated E-graph shown here is generated from the preceding rewriting phase. In Phase 1, the algorithm must select a node from `eclass2` to form the recurrence path. Choosing $n2$ results in a 4-node path (Path 1), whereas choosing $n3$ yields a shorter 3-node path (Path 2). To minimize $Cost_{cycle}$, the extractor selects $n3$. Once these cycle-critical nodes are fixed, Phase 2 resolves the remaining off-cycle dependencies, such as `eclass4`. Since neither $n5$ nor $n6$ affects the recurrence path ($C(v) = 0$), the extractor evaluates their area cost $A(v)$. It prioritizes $n6$ (Area: 2) over $n5$ (Area: 10) to minimize $Cost_{area}$. Ultimately, the extracted combination $\{n1, n3, n6, n7\}$ achieves both maximum throughput and minimal resource consumption.

3.3 DFG Mapping and Evaluation

Once the E-graph based fusion stage produces an optimized DFG, the remaining task is to map it onto the target CGRA and evaluate the resulting performance. The mapping employs a modulo scheduling algorithm over the CGRA's Modulo Routing Resource Graph [21, 22]. Starting from the theoretical MII lower bound, which is the maximum of ResMII and RecMII, the mapper iteratively attempts placement and routing at increasing II values until a valid schedule is found.

A key challenge introduced by the fusion stage is that the optimized DFG may contain arbitrary coarse-grained operators not present in a conventional FU library. To bridge this gap, we develop a parameterized template FU library that takes the set of extracted fusion patterns as input and automatically instantiates the corresponding hardware FUs. This

design decouples the fusion exploration from the architectural definition: regardless of which patterns the E-graph extraction selects, the FU library can generate matching hardware support without manual intervention, ensuring end-to-end compatibility between the compiler and the architecture. The compiled mapping results, together with the PPA characteristics of the instantiated FUs, constitute the final evaluation metrics reported in Section 4.

4 Experimental Evaluation

4.1 Setup

To assess the proposed framework, we employ benchmarks spanning multiple domains, including embedded systems, machine learning, and HPC. For embedded systems, we evaluate digital signal processing kernels, including finite impulse response filters (`fir`), normalized lattice filters (`latnrm`), fast Fourier transforms (`fft`), and dynamic time warping (`dtw`). For machine learning workloads, we use sparse matrix–vector multiplication (`spmv`), convolution (`conv`), and rectified linear unit (`relu`). HPC benchmarks include biconjugate gradient (`bicg`), general matrix-vector multiplication (`gemv`), histogram computation (`hist`), matrix–vector product with transpose (`mvt`), and generalized matrix multiplication (`gemm`). In addition, we assess full applications such as ResNet, attention mechanisms, and Harris corner detection. To improve CGRA utilization, all benchmarks are transformed through loop flattening and unrolling to generate large DFGs.

Our NUKE framework is built on the MLIR infrastructure, leveraging its modularity and powerful capabilities for dialect definition, transformation, and optimization. For the E-Graph based matching algorithms, we build on the open-source *egg* framework [25] and add approximately 2K lines of Rust code for the specialized extraction.

4.2 Main Results

Table 1 summarizes the compilation behavior on a 3×3 CGRA for different UF settings. Across all evaluated application domains, NUKE consistently reduces the total number of DFG nodes compared to the baseline compiler. This systematic graph compression validates the effectiveness of our dynamic rule synthesis in extracting and fusing compound operations. By consolidating these primitive nodes into coarse-grained operators, the framework intuitively alleviates hardware resource pressure and shortens critical recurrence paths. Consequently, NUKE achieves noticeable reductions in both the resource and recurrence minimum initiation intervals. These lowered theoretical bounds directly translate into superior final execution performance. As indicated by the blue entries, NUKE yields a strictly lower or identical Compiled II across all benchmarks. At the aggregate level, the 3×3 results correspond to about 1.28× average speedup and 1.23× average

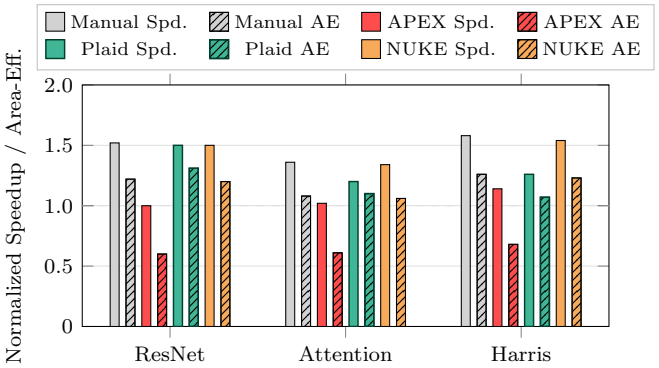


Figure 7: End-to-end comparison across full applications. Speedup and area-efficiency are normalized to the unfused baseline. NUKE, Plaid, and APEX are compared with the manual-fusion reference.

area-efficiency when the best UF is selected for each kernel. Furthermore, the performance advantages become increasingly prominent at higher loop unrolling factors like UF = 2 and UF = 4, where the baseline method frequently struggles with severe routing congestion and resource constraints.

The bound type of each kernel further modulates the optimization outcome. (1) **RecMII-bound kernels** dominates our benchmark suite at low UFs. These kernels benefit from the recurrence-aware extraction that shortens recurrence paths by fusing sequential operations within cycles. This yields consistent II reductions as the performance is primarily limited by recurrence delay rather than PE availability. However, these gains depend on whether operations in the critical cycle can be legally fused. In `latnrm`, for example, the RecMII stays high because its critical cycles contain operations that exceed single-tile PPA limits and cannot be merged. Even if NUKE shrinks the rest of the graph, these stubborn cycles prevent II improvement. As the UF increases, the main bottleneck shifts. Many kernels originally limited by recurrence delay become limited by resource availability, changing the primary goal from shortening cycles to reducing node counts. (2) **ResMII-bound kernels** benefit primarily from node count reduction, which directly lowers the ResMII and alleviates PE contention. For intrinsically ResMII-limited kernels such as `mvt`, this compaction enables II improvements even at low UFs where baseline resource pressure is moderate. At higher UFs, the expanded DFG size pushes more kernels into the ResMII-bound regime, making node reduction the dominant performance driver.

These observations confirm that our two-phase extraction strategy effectively exploits both bottleneck types to ensure that performance benefits become increasingly pronounced as DFG complexity and hardware resource pressure grow.

Domain	Kernel	UF = 1				UF = 2				UF = 4			
		#Nodes	ResMII	RecMII	CompiledII	#Nodes	ResMII	RecMII	CompiledII	#Nodes	ResMII	RecMII	CompiledII
Embedded Domain	dtw	140→129	5→5	5→5	7→7	187→160	8→7	8→6	13→10	224→192	9→8	14→10	16→10
	fft	69→65	4→4	5→5	5→5	108→88	7→6	5→4	7→6	130→106	12→10	5→4	12→10
	latnrm	73→66	7→6	13→12	17→16	93→84	8→7	13→12	18→16	132→117	12→12	13→13	19→19
Machine Learning	conv	50→44	5→4	9→6	10→8	62→56	7→6	10→8	12→10	81→68	8→6	9→7	13→11
	relu	103→88	4→4	9→7	11→10	130→116	6→5	9→7	12→10	156→139	7→–	9→7	11→9
	spmv	60→55	4→3	5→5	6→5	81→67	6→5	5→5	7→6	97→80	8→5	5→5	8→6
High-Perf. Computing	bicg	141→109	5→4	9→8	11→10	177→130	7→5	9→8	12→10	212→156	10→7	9→8	12→11
	gemm	174→152	7→6	13→11	16→14	214→186	9→7	13→11	18→15	257→223	6→5	13→11	16→14
	gemv	120→104	6→5	9→7	10→9	148→125	7→6	9→7	11→10	178→150	8→6	9→7	15→11
	hist	54→48	2→2	5→5	5→5	69→59	4→3	5→5	6→5	85→76	6→5	5→5	6→5
	mvt	145→119	6→5	5→5	8→6	185→156	8→7	5→4	9→7	222→187	13→10	5→4	13→10

Table 1: Compilation results on a 3×3 CGRA. Each cell follows the format of Baseline → NUKE where blue entries indicate an improvement in Compiled II. "–" correspond to data that are unavailable due to failed mapping.

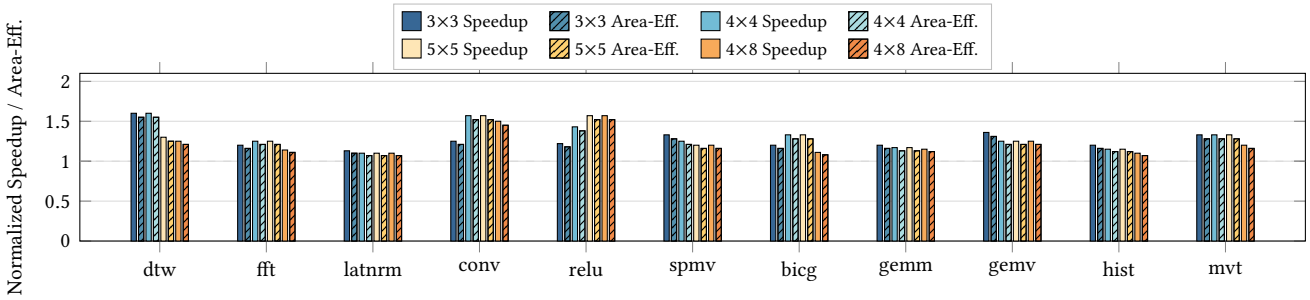


Figure 8: Scalability of NUKE across CGRA sizes. Speedup and area-efficiency are normalized to the baseline with the corresponding dimension. The best UF is selected for each individual kernel.

4.3 End-to-End Comparison

Figure 7 presents the end-to-end comparison between NUKE, APEX, Plaid, and a manual-fusion reference across ResNet, Attention, and Harris Corner detection. The unfused baseline is identical to NUKE except that pattern fusion is disabled, while the manual reference employs expert-designed, application-specific patterns. On average across the three applications, NUKE achieves 1.46× normalized speedup and 1.16× normalized area-efficiency. Among them, Harris benefits the most from fusion, ResNet remains consistently strong, and Attention exhibits a more moderate improvement.

Compared to automated baselines, NUKE delivers distinct advantages. While Plaid relies on fixed-size motifs, NUKE dynamically extracts variable-size patterns under architectural constraints, thereby achieving higher performance with competitive area-efficiency. Meanwhile, APEX fails to reach comparable levels due to its limited desin space and lack of support for complex multi-cycle operations. Furthermore, across these applications, NUKE closely approaches the manual reference in both normalized speedup and area-efficiency without requiring labor-intensive domain exploration. This success stems from its E-graph saturation alongside a two-phase extraction strategy, which effectively navigates the design space to balance performance against hardware overhead.

4.4 Scalability

Figure 8 illustrates the scalability of NUKE across four distinct array configurations, specifically 3×3, 4×4, 5×5, and 4×8. At the 3×3 scale, NUKE delivers an average speedup of about 1.28× with a corresponding average area efficiency of about 1.23× across all evaluated kernels. As the CGRA expands to 4×4 and 5×5, NUKE sustains comparable or slightly higher gains, while the 4×8 configuration remains competitive with a modestly lower average improvement. These results indicate that the proposed fusion approach scales well to larger computational fabrics, although the final gains still depend on mapper behavior.

4.5 Ablation Study

Figure 9 illustrates the effectiveness of the two-phase extraction strategy in NUKE by evaluating its performance against area-only and latency-only baselines on a 3×3 CGRA. The results in the left subplot demonstrate that an area-only approach leads to suboptimal speedup because it fails to prioritize critical recurrence paths, which often prevents the framework from effectively reducing the RecMII of the DFG. In contrast, the right subplot highlights that a latency-only strategy results in poor area-efficiency because it aggressively

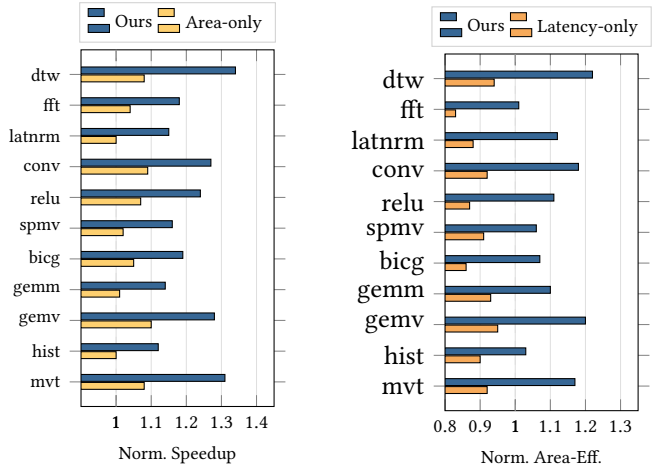


Figure 9: Ablation study of the two-stage extraction strategy on a 3×3 CGRA. Speedup and area-efficiency are normalized to the random extraction baseline. Best UF is selected per kernel.

Benchmarks	Egg Nodes	Time in MLIR (s)	Time in E-graph (s)	Time in Mapping (s)	Total Time (s)
dtw	3549	2.07	5.45	465.87	473.39
fft	1571	1.79	2.31	594.30	598.40
latnrm	1332	1.44	1.66	446.02	449.13
conv	222	0.79	2.15	6.46	9.40
relu	2135	1.45	4.58	84.86	90.89
spmv	734	1.22	1.07	74.85	77.14
bicg	3178	1.85	4.34	253.86	260.05
gemm	5723	1.89	6.53	439.82	448.25
gemv	1559	1.30	4.49	277.54	283.32
hist	534	1.12	1.68	34.36	37.15
mvt	4139	2.12	6.24	424.00	432.37

Table 2: Compilation runtime of NUKE for the evaluated kernels. Time in MLIR includes lowering and hardware merging while Time in E-graph corresponds to the saturation of root expressions.

minimizes execution cycles while ignoring the hardware overhead of the extracted patterns. By adopting a sequential optimization flow that first shortens recurrence paths and subsequently refines for area reduction, NUKE achieves a superior balance between performance and resource efficiency that avoids the distinct trade-offs inherent in single-objective methodologies. This two-stage methodology proves essential for mapping quality because it improves both RecMII and ResMII constraints simultaneously.

4.6 Runtime Analysis

Table 2 characterizes the compilation overhead for each kernel to evaluate the efficiency of the NUKE optimization pipeline across the benchmark suite. Although the E-graph saturation process generates a significant number of e-nodes for complex kernels, the total time consumed by the E-graph stage remains consistently under 7 seconds for all evaluated benchmarks. These results indicate that the proposed fusion methodology is highly efficient because the E-graph optimization constitutes only a minor fraction of the overall compilation latency. The experimental data confirms that the mapping stage remains the dominant performance bottleneck, which is consistent with prior research which suggests that the scalability of CGRA toolchains is typically constrained by the complexity of the backend mapping algorithms rather than the frontend optimization phases [28]. By maintaining a lightweight optimization stage, NUKE enables aggressive fusion exploration without significantly extending the compilation runtime.

4.7 Hardware Evaluation

To evaluate the physical overhead, we synthesize the architecture using Synopsys Design Compiler and Cadence Innovus under the ASAP 7nm PDK at 1 GHz. The 33×33 CGRA occupies 0.252 mm² with an average power of 263.7 mW. The customized template FUs incur minimal overhead, accounting for only 10.1% of per-tile area and 6.5% of power (with control and interconnect consuming 54.7% and 20.3% area, respectively), confirming the hardware efficiency of our fused units.

5 Related Works

Pattern Fusion. Pattern fusion is fundamental to achieving high CGRA efficiency. Beyond manual specialization methods [7, 8, 12–14], APEX [10] mines frequent subgraphs to automate processing element design space exploration, while Plaid [9] maps operations into fixed-size motifs with aligned compute and communication provisioning. These methods restrict fusion choices to mined subgraphs or predefined motif sizes. In contrast, NUKE dynamically constructs variable-size patterns under tile PPA constraints and retains interacting fusion alternatives in an E-graph.

E-Graph Based Optimization. Equality saturation using E-graphs provides a robust solution for complex design space exploration. This technique has been successfully adopted in logic synthesis [26, 29–31], FPGA HLS via SEER [27], and RISC-V instruction customization via ISAMORE [32].

6 Conclusion

This work proposes NUKE, an automatic pattern fusion framework that enables efficient global design space exploration for CGRAs using equality graphs. It leverages hybrid rewrite-rule generation and a two-stage extraction procedure to overcome the phase-ordering problem, effectively balancing critical path reduction with hardware efficiency. Evaluation shows that NUKE systematically optimizes fusion trade-offs, delivering significant improvements in both speedup and area-efficiency that closely approach the performance of manual design across diverse applications.

References

[1] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep convolutional neural network architecture with reconfigurable computation patterns," *TVLSI*, vol. 25, no. 8, pp. 2220–2233, 2017.

[2] S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, L. Liu, and S. Wei, "A 1.06-to-5.09 tops/w reconfigurable hybrid-neural-network processor for deep learning applications," in *VLSIC*. IEEE, 2017, pp. C26–C27.

[3] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina *et al.*, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *MICRO*, 2019, pp. 14–27.

[4] I. Bae, B. Harris, H. Min, and B. Egger, "Auto-tuning cnns for coarse-grained reconfigurable array-based accelerators," *TCAD*, vol. 37, no. 11, pp. 2301–2310, 2018.

[5] J. Lee and J. Lee, "Np-cgra: Extending cgras for efficient processing of light-weight deep neural networks," in *DATE*. IEEE, 2021, pp. 1408–1413.

[6] R. Prabhakar, R. Sivaramakrishnan, D. Gandhi, Y. Du, M. Wang, X. Song, K. Zhang, T. Gao, A. Wang, X. Li *et al.*, "Sambanova sn40l: Scaling the ai memory wall with dataflow and composition of experts," in *MICRO*. IEEE, 2024, pp. 1353–1366.

[7] Y. Luo, C. Tan, N. B. Agostini, A. Li, A. Tumeo, N. Dave, and T. Geng, "Ml-cgra: an integrated compilation framework to enable efficient machine learning acceleration on cgras," in *DAC*. IEEE, 2023, pp. 1–6.

[8] J. Qin, T. Xia, C. Tan, J. Zhang, and S. Q. Zhang, "Picachu: Plug-in cgra handling upcoming nonlinear operations in llms," in *ASPLOS*, 2025, pp. 845–861.

[9] Z. Li, P. Dangi, C. Yin, T. K. Bandara, R. Juneja, C. Tan, Z. Bai, and T. Mitra, "Enhancing cgra efficiency through aligned compute and communication provisioning," in *ASPLOS*, 2025, pp. 410–425.

[10] J. Melchert, K. Feng, C. Donovick, R. Daly, R. Sharma, C. Barrett, M. A. Horowitz, P. Hanrahan, and P. Raina, "Apex: A framework for automated processing element design space exploration using frequent subgraph analysis," in *ASPLOS*, 2023, pp. 33–45.

[11] G. Charitopoulos, D. N. Pnevmatikatos, and G. Gaydadjiev, "Mc-def: Creating customized cgras for dataflow applications," *TACO*, vol. 18, no. 3, pp. 1–25, 2021.

[12] M. Jiang, Y. Zhu, H. You, C. Tan, Z. Li, J. Xu, and L. Ju, "Fhe-cgra: Enable efficient acceleration of fully homomorphic encryption on cgras," in *DAC*, 2024, pp. 1–6.

[13] K. Koul, J. Melchert, K. Sreedhar, L. Truong, G. Nyengele, K. Zhang, Q. Liu, J. Setter, P.-H. Chen, Y. Mei *et al.*, "Aha: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 2, pp. 1–34, 2023.

[14] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, "Riptide: A programmable, energy-minimal dataflow compiler and architecture," in *MICRO*. IEEE, 2022, pp. 546–564.

[15] S. Liu, J. Weng, D. Kupsh, A. Sohrabizadeh, Z. Wang, L. Guo, J. Liu, M. Zhulin, R. Mani, L. Zhang *et al.*, "Overgen: Improving fpga usability through domain-specific overlay generation," in *MICRO*. IEEE, 2022, pp. 35–56.

[16] O. Ragheb, S. Wicklund, M. Walker, R. Beidas, A. Ragab, T. Yu, and J. Anderson, "Cgra-me 2.0: A research framework for next-generation cgra architectures and cad," in *IPDPSW*. IEEE, 2024, pp. 642–649.

[17] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, 2000.

[18] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "Piperench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[19] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[20] J. D. Souza, L. Carro, M. B. Rutzig, and A. C. S. Beck, "A reconfigurable heterogeneous multicore with a homogeneous isa," in *DATE*. IEEE, 2016, pp. 1598–1603.

[21] B. R. Rau, "Iterative modulo scheduling," *Int. J. Parallel Program.*, vol. 24, no. 1, pp. 3–64, 1996.

[22] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *DATE*, 2003, pp. 296–301.

[23] C. G. Nelson, *Techniques for program verification*. Stanford University, 1980.

[24] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: a new approach to optimization," in *POPL*, 2009, pp. 264–276.

[25] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *PACMPL*, vol. 5, no. POPL, pp. 1–29, 2021.

[26] C. Chen, G. Hu, C. Yu, Y. Ma, and H. Zhang, "E-morphic: Scalable equality saturation for structural exploration in logic synthesis," in *DAC*. IEEE, 2025, pp. 1–7.

[27] J. Cheng, S. Coward, L. Chelini, R. Barbalho, and T. Drane, "Seer: Super-optimization explorer for high-level synthesis using e-graph rewriting," in *ASPLOS*, 2024, pp. 1029–1044.

[28] M. Balasubramanian and A. Shrivastava, "Pathseeker: a fast mapping algorithm for cgras," in *DATE*. IEEE, 2022, pp. 268–273.

[29] S. Coward, G. A. Constantinides, and T. Drane, "Automating constraint-aware datapath optimization using e-graphs," in *DAC*. IEEE, 2023, pp. 1–6.

[30] C. Chen, G. Hu, D. Zuo, C. Yu, Y. Ma, and H. Zhang, "E-syn: E-graph rewriting with technology-aware cost functions for logic synthesis," in *DAC*, 2024, pp. 1–6.

[31] J. Yin, Z. Song, C. Chen, Y. Cai, Z. Zhang, and C. Yu, "e-boost: Boosted e-graph extraction with adaptive heuristics and exact solving," in *ICCAD*. IEEE, 2025, pp. 1–9.

[32] Y. Xiao, C. Yin, Y. Sun, Y. Zou, and Y. Liang, "Finding reusable instructions via e-graph anti-unification," in *ASPLOS*, 2026, pp. 749–763.