Beyond Surface-Level Pattern Trap: LLM Agents for Faster and Smarter Cross-Architecture Code Migration

Weijia Li[1,2], Ke Gao[1,2], Pengfei Chen[1,2], Jiajie Li[1,2], Xinyu Wang[1,2], Yiran Le[1,2], Yize Wu[1,2], Ling Li [1,2]*

[1]Intelligent Software Research Center, Institute of Software, CAS, Beijing, China
[2]University of Chinese Academy of Sciences, Beijing, China

liweijia2025@iscas.ac.cn, liling@iscas.ac.cn

Abstract

The problem of surface-level pattern mapping represents a critical yet underexplored failure mode in large language model (LLM) reasoning, and is particularly acute in cross-architecture code migration of high-performance libraries. On low-resource, low-level code, insufficient coverage in pretraining data often leads LLMs to rely on superficial name- or type-based correspondences, rather than principled refactorization and reasoning grounded in core functional semantics and architecture-specific optimization intents. This tendency severely hampers the effectiveness of LLMs in complex migration scenarios. To address these challenges, we propose FSCM, a multi-agent framework for cross-architecture migration. FSCM decouples complex implementation details through functional mining and code refactoring, guiding LLMs to focus on invariant semantic anchors across architectures. By mitigating surface-level pattern traps, FSCM improves both functional correctness and performance when targeting emerging architectures. Extensive experiments on the challenging real-world OpenCV library migration tasks demonstrate substantial improvements over state-of-the-art baselines, achieving up to 22% higher correctness rates over Copilot and 43.04× speedup on RISC-V platforms. Code and data are available at: https://github.com/breezejh/FSCM.

1 Introduction

With the booming development of heterogeneous architecture, cross-architecture code migration has emerged as an important research problem. It refers to adapting source code designed for one hardware architecture to another, such as ARM-to-RISC-V migration, thereby improving the flexibility and sustainability of architecture-specific ecosystems. Among various migration scenarios, high-performance libraries are particularly critical and challenging due to the prevalence and complexity. Effective migration requires not only functional correctness, but also full utilization of the computational resources of the target architecture to maximize the target's performance.

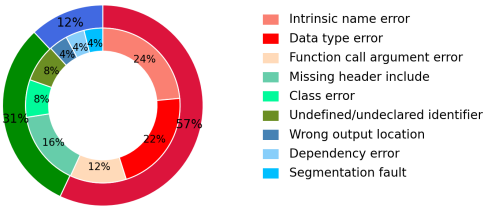

Figure 1: Proportions of LLM-induced errors. Red indicates errors directly caused by surface-level pattern matching, while green denotes indirect effects.

Despite its importance, cross-architecture code migration remains a resource-intensive and error-prone task. Manual migration is excessively costly, and rule-based tools provide only limited coverage, underscoring the urgent need for efficient and reliable automated solutions.

Recent advancements in LLMs have opened new approaches for this task. However, the imbalanced distribution of training code corpora causes models to overfit high-resource code (such as Python) while learning only surface-level patterns for low-resource code (such as Intrinsics), making it difficult for them to grasp deeper structures and semantics. Moreover, LLM inference highly depends on the patterns present in the context. When exposed to source-architecture code or documentation as contextual input, the model tends to mimic surface-level patterns, such as syntax templates and naming conventions, as illustrated in Figure 1. A detailed case study is provided in Figure 2.

Some existing approaches(Aycock et al., 2025; Zhang et al., 2025a) address data sparsity by fine-tuning LLMs on synthesized parallel code data, but

*Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 3029–3042
July 2-7, 2026

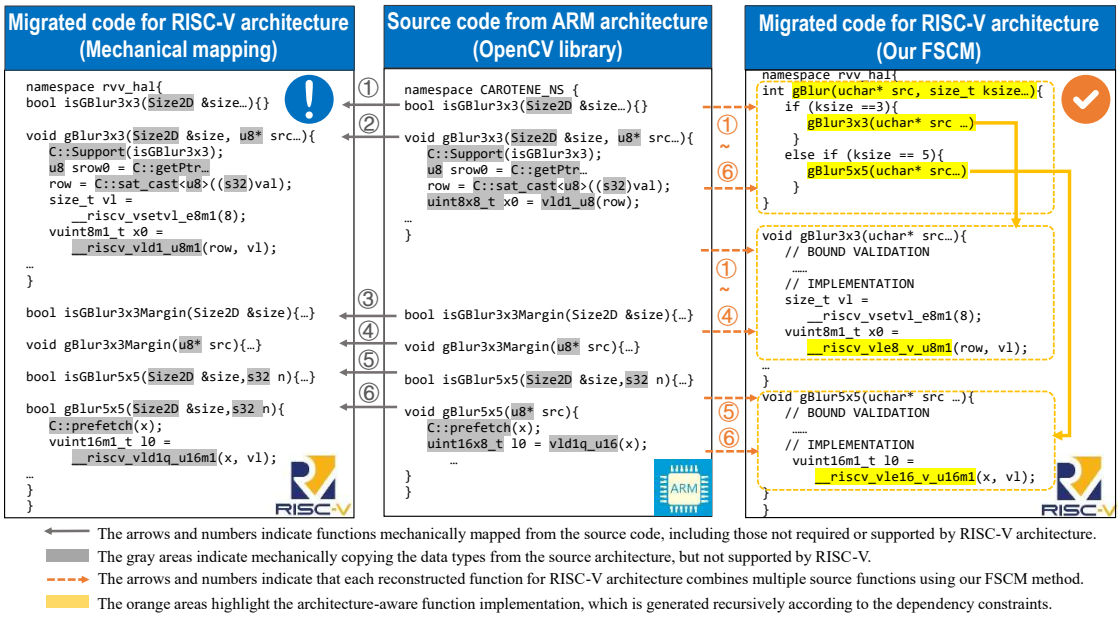

Figure 2: Illustration of the challenges and method comparisons for cross-architecture migration tasks.

they are only effective at learning local instruction-level mappings. Other approaches build program skeletons(Wang et al., 2025; Zhang et al., 2025b) as intermediate representations to reduce translation difficulty, but they are restricted to line- or function-level granularity of source code, which is still a surface-level mapping.

Considering these limitations, we argue that the effective semantic information for code migration is inherently deep, structurally complex, and requires long-range reasoning across functions or files to be correctly understood. To address these challenges, we propose FSCM, a novel multi-agent framework for automatic cross-architecture code migration. FSCM refactors highly architecture-specific source code in a deep, semantics-oriented manner, explicitly decoupling program semantics from low-level implementation details. Building on this, FSCM enables function-aware, functionally equivalent migration and architecture-aware code performance optimization, thereby improving both correctness and performance in complex migration tasks.

Our contributions are summarized as follows:

- We identify a key limitation of LLMs in low-resource code migration: they often rely on surface-level pattern matching rather than deep semantic understanding.
- Building on this insight, we propose FSCM, a framework that orchestrates agents for code refactoring, semantic understanding, and multi-stage testing. FSCM decouples complex implementation details through functional mining and code refactoring, guiding LLMs to focus on invariant semantic anchors across architectures.
- Extensive experiments on the challenging real-world task of ARM-to-RISC-V migration demonstrate substantial improvements over state-of-the-art code intelligence agents.

2 Preliminary

2.1 Architecture-Specific Code and Intrinsics

On modern processors, data-level parallelism is a key mechanism for accelerating compute-intensive workloads in high-performance libraries. Among existing techniques, compiler auto-vectorization requires no code modification but often yields marginal or degraded performance due to conservative or inappropriate decisions. Handwritten assembly offers strong control but lacks program semantics, losing the ability for compositional optimizations. Therefore, many high-performance libraries adopt SIMD intrinsics to optimize algorithms. Intrinsics directly map to hardware SIMD instructions, and their effective use requires careful handling of issues such as register allocation and memory access. Developers often refer to existing architecture implementations when generating intrinsic code for heterogeneous architectures.

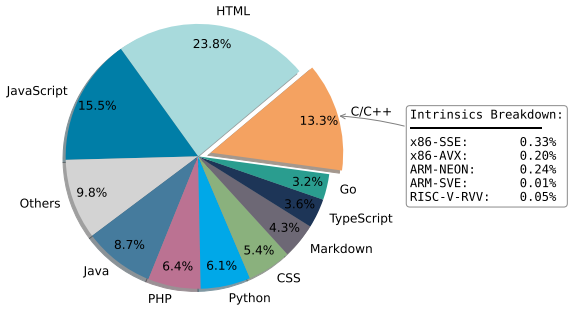

Figure 3: Distribution of the Stack v1 Dataset.

2.2 High-Resource and Low-Resource Code

In data-driven code generation scenarios, there exists a significant disparity in the coverage of different code within training corpora. The Stack v1 (Kocetkov et al., 2022) constitutes one of the primary code data sources for most pre-trained models. Its data distribution is presented in Figure 3. We define high-resource code as code that appears frequently in public code repositories and training datasets, with abundant examples and broad coverage, such as HTML, JavaScript, and general C/C++ implementations. In contrast, low-resource code is sparse, fragmented, and underrepresented in training datasets. It often involves domain-specific semantics and long-tail constraints. SIMD intrinsics are a typical case: their semantics depend heavily on hardware architecture and compiler implementation, and relevant knowledge is scattered across hardware manuals, technical documents, and hands-on optimization practice.

2.3 Deep Semantics vs. Surface-level Pattern

The core challenge of cross-architecture migration lies in distinguishing deep semantics from surface-level implementation patterns. Surface-level pattern matching focuses on syntactic and local structural similarities—such as specific instruction sequences, function organization, or naming conventions. These patterns are often tightly coupled with particular architectures and historical implementations. Directly transplanting them can easily lead to incompatibilities with the target architecture.

In contrast, deep semantics refer to the abstract computational logic, data dependencies, and optimization intent encoded in a program, which determine its functional behavior and performance objectives and are largely transferable across architectures.

3 Methodology

In this section, we introduce the FSCM framework, as illustrated in Figure 4.

3.1 Code Refactoring

The original dependency graph of the repository is extremely complex, entangling legacy code, duplicated logic, and numerous fragmented helper functions. Directly leveraging such a dependency graph leads to lengthy and distracting contextual information, which is detrimental to low-resource code migration. Achieving a clear and hierarchical new code structure through refactoring, without altering overall behavior, is a prerequisite to successful migration.

We implemented a suite of C++ static analysis tools, including global parsing and code refactoring. In the global parsing phase, the analyzer recursively traverses the directory hierarchy, identifying syntax-level entities and their interconnections. In this process, the parser skips built artifacts and version control files, preliminarily filtering out non-functional entities. The constructed graph is denoted by $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{T}, \mathcal{R})$ where $\mathcal{G}$ is a directed graph, $\mathcal{V}$ is the set of nodes, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges. For each node $v \in \mathcal{V}$, the mapping $\tau(v) : \mathcal{V} \to \mathcal{T}$ maps the node to a type $t \in \mathcal{T}$ where $\mathcal{T} = \{directory, file, class, struct, function\}$. For each edge $e \in \mathcal{E}$, the mapping $\rho(e) : \mathcal{E} \to \mathcal{R}$ maps the edge to a relationship $r \in \mathcal{R}$ where $\mathcal{R} = \{contains, invokes, inherits, imports\}$. Due to the complexity of C++, the parser performs additional steps detailed in Appendix A.1.

In the code refactoring phase, the global dependency graph serves as the basis for pruning and merging operations. We iteratively apply a prune-and-merge strategy that contracts structurally insignificant nodes into their invokers, while preserving nodes that are critical to semantic integrity and architectural organization. The pruning process is guided by structural degree analysis: functions with low in-degree typically represent leaf-level implementation details and contribute little independent semantic value. Such nodes are candidates for merging, provided that their integration does not violate semantic traceability or architectural boundaries. To prevent over-aggressive abstraction, we introduce a structural importance assessment that explicitly preserves nodes with high fan-out, inheritance relationships, or cross-module import

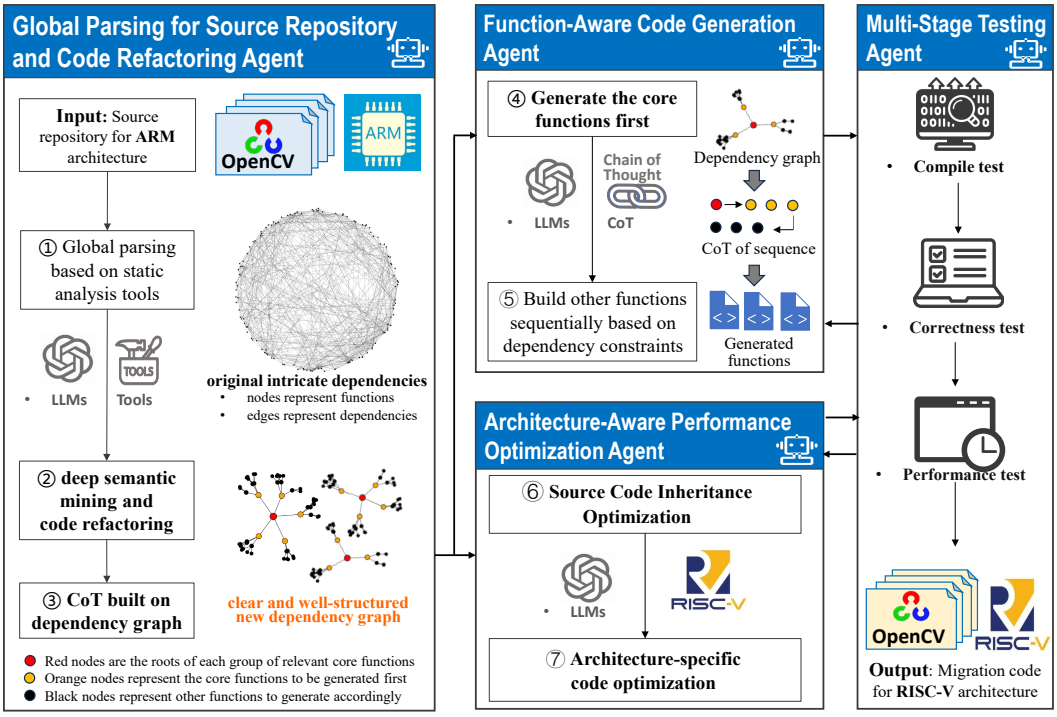

Figure 4: Overview of the FSCM framework. It guides LLM agents to first refactor the source code and then generate the required functions, significantly improving the efficiency and performance of complex migration tasks.

significance. Details in Appendix A.1.

Through iteration, the global graph is transformed into a reduced yet semantically faithful abstraction. This process shrinks the graph's size by at least 30% while preserving all critical architectural relationships, substantially simplifying downstream code generation.

3.2 Semantic-Aware Code Generation

This subsection focuses on deep-semantic code generation based on the reconstructed code dependency graph. Deep semantics are characterized along two dimensions: functional equivalence and performance optimization.

Function-Aware Code Generation

The first step focuses on migrating code from the source architecture to a simple, target-agnostic scalar implementation. The primary goal of this stage is to establish functional equivalence, ensuring that the fundamental logic of the operator is correctly translated.

Guided by Chain-of-Thought (CoT) representations, the agent analyzes the vectorization-independent semantics of each node—including control flow and boundary handling—and generates a baseline scalar C/C++ implementation. Depending on semantic complexity, it selectively inspects a function's internal implementation to extract essential algorithmic behavior while avoiding low-level details. After validation with the correctness-test tool, the scalar code acts as a verified semantic anchor, ensuring functional correctness and providing a foundation for subsequent optimizations.

Architecture-Aware Performance Optimization

FSCM performs architecture-aware performance optimization to bridge the performance gap induced by architectural and instruction-set mismatches. FSCM explicitly separates performance optimization into two steps: (1) optimization inheritance from the source code, and (2) architecture-specific optimization knowledge injection. This design allows FSCM to preserve the optimization logic from the source code while fully exploiting the capabilities of target architecture.

Step1: Optimization Inheritance from Source Code.

High-performance source libraries encapsulate rich optimization knowledge beyond architecture-specific intrinsics, including vectorization-friendly loop structures, data layout decisions, and algorithmic fast paths. FSCM captures these optimization intents through CoT analysis at each node and leverages them to guide the generation from scalar

implementations toward vectorized optimized code, thereby avoiding the structural performance degradation commonly observed in direct LLM-based translation.

Step2:Architecture-Specific Optimization Injection.

Subsequently, an Architecture-Aware Optimization Agent injects performance strategies tailored to the target architecture. For RISC-V Vector Extension (RVV), this includes vector-length-agnostic programming, register grouping, and semantically correct intrinsic selection. The agent learns the target architecture's optimization strategies from a single prompt and adapts its optimization decisions while respecting the constraints established in Step 1. The optimized code is iteratively validated through compilation tests, enabling progressive refinement.

4 Experiment Setup

4.1 Hardware Platforms

To evaluate the generality of FSCM across heterogeneous RISC-V platforms, we select two representative processors as our target platforms: SpacemiT X60[1] and Alibaba T-Head C908[2].

4.2 Datasets

We chose OpenCV as our primary benchmark because of its broad adoption from desktop to embedded and IoT scenarios. Our dataset focuses on the performance hotspots ImgProc and Core modules, with ARM Neon-optimized code suitable for cross-architecture migration. Details are in the Appendix A.2.

4.3 Baselines

Skeleton-Based Approach. We include the skeleton-based approach (Zhang et al., 2025b) as a representative program migration method, which uses the source code's class and function structure as a "skeleton" to guide code generation.

Multi-Agent. We adopt the state-of-the-art multi-agent frameworks `CodeX` (OpenAI, 2025) and `Copilot` (GitHub, 2025) as baselines. In particular, `Copilot` integrates with the IDE and uses compiler errors as feedback to iteratively refine generated code until it compiles successfully.

[1]https://www.spacemit.com/en/spacemit-x60-core/

[2]https://www.xrvm.cn/product/xuantie/C908

4.4 Evaluation Metrics

To evaluate cross-architecture code migration, we report three complementary metrics:

(1) **CS**: a binary indicator of whether each migrated function compiles; CS acts as a gate for later evaluation.

(2) **TS**: the fraction of OpenCV official test cases passed by the migrated code.

(3) **SR**: $T_{\text{scalar}}/T_{\text{vector}}$, comparing the scalar baseline against the migrated vectorized version.

Formal definitions and implementation details are provided in Appendix A.5.

4.5 Implementation

We select `claude-sonnet-4-20250514` (Anthropic, 2024) and `Grok-4`(xAI, 2025) as the base models for our method and the comparative methods. For each case in our dataset, we provide the source ARM Neon implementation to each method and request migration to RISC-V vector extensions.

5 Results

5.1 Migration Accuracy

We evaluate the functional correctness by measuring its ability to generate code that successfully compiles and passes OpenCV's official test suites.

Results. Table 1 presents the compilation status (CS) and test success rate (TS) for OpenCV dataset. FSCM achieves significantly higher correct coverage compared to all baseline methods. On the OpenCV dataset, A key observation is that naive LLM migration frequently fails, largely due to surface-level pattern matching. The function-level Skeleton approach provides limited relief in the simple module Core, but remains ineffective in the complex module Imgproc, where deep inter-function dependencies exceed the expressiveness of function-level skeletons.

Among state-of-the-art code intelligence agents, Copilot achieves competitive performance, benefiting from multi-turn interaction and tool calling. Even though Copilot can iteratively incorporate feedback until the code compiles, that feedback is largely syntactic and local. Without explicitly modeling deeper functional semantics, these feedback-driven iterations often produce superficial fixes that satisfy compilation constraints but fail to preserve correct behavior. Consequently, FSCM achieves a test success rate of 78%, substantially outperforming the best baseline, Copilot (59%).

Table 1: Performance comparison across different migration methods, LLMs, and OpenCV operators for ARM Neon to RISC-V vector migration. ✓ indicates success; × indicates failure. Numbers in parentheses show test pass rates (passed/total)."/" indicates not applicable due to compilation failure.

Function	Baseline Methods								Ours			
	Claude4 (Anthro. 2025)		Skeleton (2025)		CodeX Agent (OpenAI 2025)		Copilot Agent (GitHub 2025)		+Grok4		+Claude4	
	CS	TS	CS	TS	CS	TS	CS	TS	CS	TS	CS	TS
Imgproc												
box_filter	✓	✓(328/328)	×	/	×	/	✓	✓(328/328)	✓	✓(328/328)	✓	✓(328/328)
colorconvert	×	/	×	/	×	/	✓	✓(813/813)	✓	×(717/813)	✓	×(717/813)
convolution	×	/	×	/	×	/	✓	✓(2400/2400)	✓	✓(2400/2400)	✓	✓(2400/2400)
gaussian_blur	×	/	×	/	×	/	✓	×(2/3)	✓	✓(3/3)	✓	✓(3/3)
integral	×	/	✓	×(16/17)	✓	×(16/17)	✓	×(16/17)	✓	✓(17/17)	✓	×(16/17)
median_filter	×	/	×	/	✓	×(62/66)	✓	×(65/66)	✓	✓(66/66)	✓	✓(66/66)
morph	×	/	×	/	✓	×(553/555)	✓	×(514/555)	✓	✓(555/555)	✓	×(553/555)
pyrdown	×	/	×	/	×	/	✓	×(117/121)	✓	✓(121/121)	✓	✓(121/121)
pyrup	×	/	×	/	×	/	✓	×(44/46)	✓	×(44/46)	✓	×(44/46)
resize_area	×	/	×	/	×	/	✓	×(144/145)	✓	×(144/145)	✓	×(144/145)
resize_linear	×	/	×	/	×	/	✓	✓(1/1)	✓	✓(1/1)	✓	✓(1/1)
resize_nearest	×	/	×	/	×	/	✓	✓(1/1)	✓	✓(1/1)	✓	✓(1/1)
sep_filter	×	/	×	/	×	/	✓	✓(256/256)	✓	✓(256/256)	✓	✓(256/256)
sobel	×	/	×	/	×	/	✓	×(609/611)	✓	×(609/611)	✓	×(609/611)
Imgproc Average	**7%**	**7%**	**7%**	**0%**	**21%**	**0%**	**100%**	**43%**	**100%**	**71%**	**100%**	**57%**
Core												
absdiff	✓	✓(114/114)	✓	✓(114/114)	✓	×(112/114)	✓	×(113/114)	✓	✓(114/114)	✓	✓(114/114)
add	×	/	✓	×(212/226)	✓	×(225/226)	✓	×(225/226)	✓	✓(226/226)	✓	✓(226/226)
add_weighted	×	/	×	/	×	/	✓	✓(114/114)	✓	✓(114/114)	✓	✓(114/114)
bitwise_and	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)
bitwise_not	✓	✓(56/56)	✓	✓(56/56)	✓	✓(56/56)	✓	✓(56/56)	✓	✓(56/56)	✓	✓(56/56)
bitwise_or	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)
bitwise_xor	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)	✓	✓(224/224)
cmp	×	/	✓	✓(169/169)	×	/	✓	✓(169/169)	✓	✓(169/169)	✓	✓(169/169)
div	✓	×(368/472)	×	/	×	/	✓	✓(472/472)	✓	×(424/472)	✓	×(471/472)
dot_prod	×	/	×	/	×	/	✓	×(56/57)	✓	✓(57/57)	✓	✓(57/57)
flip	✓	✓(169/169)	×	/	✓	×(135/169)	✓	×(168/169)	✓	×(168/169)	✓	✓(169/169)
magnitude	×	/	✓	✓(17/17)	✓	✓(17/17)	✓	✓(17/17)	✓	✓(17/17)	✓	✓(17/17)
max	✓	✓(58/58)	✓	✓(58/58)	✓	✓(58/58)	✓	✓(58/58)	✓	✓(58/58)	✓	✓(58/58)
min	✓	✓(58/58)	✓	✓(58/58)	✓	✓(58/58)	✓	✓(58/58)	✓	✓(58/58)	✓	✓(58/58)
mul	×	/	×	/	×	/	✓	✓(225/225)	✓	✓(225/225)	✓	✓(225/225)
norm	×	/	×	/	×	/	✓	×(224/261)	✓	×(260/261)	✓	✓(261/261)
sub	×	/	✓	✓(2786/2786)	×	/	✓	✓(2786/2786)	✓	✓(2786/2786)	✓	✓(2786/2786)
sum	×	/	×	/	×	/	✓	✓(56/56)	✓	✓(56/56)	✓	✓(56/56)
Core Average	**50%**	**44%**	**61%**	**56%**	**56%**	**39%**	**100%**	**72%**	**100%**	**83%**	**100%**	**94%**
Overall Average	**31%**	**28%**	**38%**	**31%**	**41%**	**22%**	**100%**	**59%**	**100%**	**78%**	**100%**	**78%**

5.2 Performance Optimization

To evaluate performance in cross-architecture migration tasks, we conduct comprehensive experiments across two RISC-V platforms: SpacemiT X60 and Alibaba T-Head C908. The evaluation focuses on runtime performance, measured as speedup relative to scalar implementations. We assess performance across multiple case studies and matrix sizes, concentrating on computationally intensive operations that benefit most from vectorization. We compare FSCM against several baseline approaches, including compiler auto-vectorization (O3), expert-optimized implementations, CodeX, and Copilot.

Results. Table 2 presents the performance comparison across different platforms and methods. FSCM consistently achieves substantial speedups, significantly outperforming compiler auto-vectorization (O3), state-of-the-art LLM agents (CodeX and Copilot), and even expert hand-optimized implementations in many cases.

Compiler auto-vectorization delivers only modest speedups. This limitation stems from conservative vectorization heuristics and the compiler's inability to restructure computation patterns that require cross-loop or cross-function reasoning.

Agents such as CodeX and Copilot exhibit highly unstable performance behavior. CodeX fre-

Table 2: Performance comparison across different RISC-V and ARM platforms, LLMs, operators, and matrix sizes. Speedup over scalar implementations (×). "*" indicates results with Architecture-specific Optimization. "failed-c" indicates compilation failure; "failed-t" indicates functional test failure. Additional results are provided in the Appendix (See Table 5). Gray background highlights the best performance.

Target Platform	Method	Add		Sub		Max		Mul	
		1280x7200 8SC1	1280x720 8UC1	640x480 8UC4	1280x720 8UC1	640x480 8UC1	640x480 8UC3	1280x720 8UC1	1280x720 8UC4
X60 (RISC-V)	Compiler-O3	7.92×	13.10×	11.80×	11.86×	10.01×	10.12×	4.64×	4.62×
	Codex	failed-t	failed-t	failed-c	failed-c	85.12×	76.28×	failed-c	failed-c
	Copilot	failed-t	failed-t	100.45×	102.15×	90.80×	40.38×	61.25×	58.41×
	Human Expert	63.87×	50.90×	121.00×	124.50×	85.12×	74.89×	59.69×	55.14×
	Ours								
	+Grok 4	80.62×	52.93×	96.80×	102.15×	71.68×	73.55×	21.06×	20.56×
	+Grok 4 *	80.62×	88.22×	102.38×	102.15×	85.12×	76.28×	21.50×	20.56×
	+Claude Sonnet 4	65.57×	69.65×	133.10×	132.80×	90.80×	76.28×	78.10×	82.65×
	+Claude Sonnet 4 *	106.91×	79.40×	133.10×	132.80×	90.80×	79.21×	88.42×	85.68×
	relative improvement	**(↑43.04×)**	**(↑28.50×)**	**(↑12.10×)**	**(↑8.30×)**	**(↑5.68×)**	**(↑4.32×)**	**(↑28.73×)**	**(↑30.54×)**

Target Platform	Method	Sub		AddWeighted		Mul		Resize	
		1920x1080 8SC1	1280x720 8SC1	640x480 8UC4	640x480 8UC1	640x480 32FC1	1920x1080 32FC1	640x480 16UC1	640x480 8UC4
C908 (RISC-V)	Compiler-O3	7.73×	7.78×	4.16×	4.18×	8.98×	9.10×	6.40×	5.78×
	Codex	failed-c	failed-c	failed-c	failed-c	failed-c	failed-c	failed-t	failed-t
	Copilot	92.96×	91.93×	26.19×	25.58×	16.35×	16.63×	failed-t	failed-t
	Human Expert	116.92×	90.29×	26.29×	25.40×	16.35×	16.65×	3.84×	2.94×
	Ours								
	+Grok 4	92.96×	91.93×	29.73×	29.29×	28.28×	28.66×	6.19×	6.25×
	+Grok 4 *	92.96×	91.93×	29.73×	29.53×	28.28×	28.66×	6.19×	6.25×
	+Claude Sonnet 4	119.38×	101.12×	31.26×	31.58×	26.83×	27.18×	0.91	0.61
	+Claude Sonnet 4 *	119.38×	114.91×	31.26×	31.58×	26.83×	27.18×	0.91×	0.61×
	relative improvement	**(↑2.46×)**	**(↑24.62×)**	**(↑4.97×)**	**(↑6.18×)**	**(↑10.48×)**	**(↑12.01×)**	**(↑2.35×)**	**(↑3.31×)**

quently fails to compile or produce functionally correct implementations. Copilot, while capable of generating vectorized code for some operators, shows large variance across different kernels and input sizes. This inconsistency suggests that existing agents primarily rely on surface-level instruction patterns or local compiler feedback, which is insufficient for systematically exploiting architectural parallelism.

This gap highlights the importance of FSCM's code refactoring. By reconstructing cross-function and cross-file dependencies, FSCM enables the model to identify more vectorization opportunities.

5.3 Ablation Study

This ablation study evaluates whether the code refactoring strategy provides benefits independent of downstream agents. To control for instruction- and optimization-related confounding factors, we directly compare scalar C++ code generated from ARM source implementations.

Experimental Setup. We design two experimental conditions: (1) *Naive CoT Generation*: LLM directly translates ARM Neon code to scalar implementations using a naive CoT; (2) *Our CoT Generation*: The same LLM generates scalar code using our refactored dependency graph as intermediate representation.

No compiler feedback or iterative repair mechanisms are employed in either setting, ensuring that the only difference between the two conditions is the code refactoring strategy.

Results. Table 3 presents the CS and TS for both approaches across OpenCV's Core and Imgproc modules. We observe that our method yields relatively modest improvements on the Core module, while producing substantial gains on the more complex Imgproc module. This stems from variations in structural complexity and dependency depth. The Core module mainly consists of low-level utilities with shallow call hierarchies, where naive generation is often sufficient. In contrast,

Table 3: Ablation study comparing direct generation vs. dependency-guided generation across OpenCV. ✓ indicates success; × indicates failure. Numbers in parentheses show test pass rates (passed/total). "/" indicates not applicable due to compilation failure.

Module Function	Grok4 + *Naive CoT*		Grok4 + *Our CoT*		Module Function	Grok4 + *Naive CoT*		Grok4 + *Our CoT*	
	CS	TS	CS	TS		CS	TS	CS	TS
Imgproc					**Core**				
box_filter	×	/	✓	✓(328/328)	absdiff	✓	×(97/114)	✓	✓(114/114)
colorconvert	×	/	✓	×(717/813)	add	✓	✓(226/226)	✓	✓(226/226)
convolution	×	/	✓	✓(2400/2400)	add_weighted	✓	✓(114/114)	✓	✓(114/114)
gaussian_blur	×	/	✓	✓(3/3)	bitwise_and	✓	✓(224/224)	✓	✓(224/224)
integral	×	/	✓	✓(17/17)	bitwise_not	✓	✓(224/224)	✓	✓(224/224)
median_filter	×	/	✓	✓(66/66)	bitwise_or	✓	✓(224/224)	✓	✓(224/224)
morph	×	/	×	/	bitwise_xor	✓	✓(56/56)	✓	✓(56/56)
pyrdown	×	/	✓	✓(121/121)	cmp	✓	✓(169/169)	✓	✓(169/169)
pyrup	×	/	✓	✓(46/46)	div	✓	×(357/472)	✓	✓(472/472)
resize_area	×	/	✓	×(144/145)	dot_prod	✓	✓(57/57)	✓	✓(57/57)
resize_linear	×	/	✓	✓(1/1)	flip	✓	✓(169/169)	✓	✓(169/169)
resize_nearest	×	/	✓	✓(1/1)	magnitude	✓	✓(17/17)	✓	✓(17/17)
sep_filter	✓	✓(256/256)	✓	✓(256/256)	max	✓	✓(58/58)	✓	✓(58/58)
sobel	✓	×(610/611)	✓	✓(611/611)	min	✓	✓(58/58)	✓	✓(58/58)
					mul	✓	×(190/225)	✓	✓(225/225)
					norm	×	/	✓	×(782/783)
					sub	×	/	✓	✓(2786/2786)
					sum	✓	✓(57/57)	✓	✓(57/57)
Average	**14%**	**7%**	**93%**	**79%**	**Average**	**89%**	**72%**	**100%**	**94%**

CS: Compilation Status; **TS**: Test Success Rate.

Table 4: Cost Summary and Comparison.

FSCM	Time	API calls	Money	I/O Token
Refactoring	1min	1	0.06	1k/0.5k
Analysis node	3min	5	0.93	14k/6k
Generation	7min	6	0.89	33k/24k
Sum	11min	12	1.88	48k/30k
Copilot	25 min	– (closed source)		
CodeX	30 min	31	$14	121k / 11k

Experiments are based on claude-sonnet-4-20250514.

Imgproc contains composite operators with deep and intertwined dependencies, in which naïve generation frequently suffers from missing or inconsistent dependencies. By explicitly modeling these dependencies, dependency graph reconstruction effectively reduces such structural errors, leading to significantly larger improvements on Imgproc.

5.4 Cost Analysis.

As shown in Table 4, FSCM introduces a code refactoring strategy to reconstruct the dependency graph, effectively narrowing the agent's search space from the entire codebase. Therefore, FSCM significantly reduces the overall time and token cost compared to other methods.

6 Related Work

Cross-Architecture Migration. Cross-Architecture code migration remains a resource-intensive and error-prone task. Consequently, manual migration typically requires expertise and scrutiny, which is costly and error-prone. Rule-based methods (Baker, 1996; Bravenboer et al., 2008; Cordy, 2006; Feldman, 1990; Klint et al., 2009) rely on manually defined rules and program analysis, such as SIMDe(Xu et al., 2023) and neon2rvv(howjmay, 2023), to transform architecture-specific constructs into their counterparts on the target ISA. However, they often rely on limited predefined rules or template matching, offering marginal optimization space and failing to scale to large codebases.

LLM for Code Migration. Prior work exploits LLMs to capture translation patterns from data(Zheng et al., 2023). Examples include migrating to Python (Pan et al., 2023), Rust (Eniser et al., 2024), or JavaScript with skeleton-based techniques (Wang et al., 2025; Zhang et al., 2025b). Beyond standalone models, LLM agents(Dong et al., 2024; GitHub, 2025; Luo et al., 2025; OpenAI, 2025; Qian et al., 2023; Sapkota et al., 2025) extend

LLM capabilities with planning, execution, and verification. Overall, the task remains underexplored, as existing methods often assume direct function-level mapping, which fails when restructuring and architecture-aware optimization are required.

7 Conclusion

This paper takes cross-architecture code migration as a representative scenario to reveal a key limitation of current LLMs: When dealing with low-resource code, LLMs often default to surface-level pattern imitation rather than reasoning about deep semantics. To address this, we propose FSCM, a multi-agent framework that performs code refactoring and suppress misleading implementation artifacts, enabling more reliable generation. Extensive experiments demonstrate substantial improvements over state-of-the-art baselines, achieving up to 22% higher correctness rates compared to the current mechanical migration strategies and a 43.04× speedup on RISC-V platforms.

8 Limitations

Domain Scope. Our experiments are limited to two C++ libraries, OpenCV and NCNN, both in the domain of scientific computing and signal/image processing. The effectiveness of FSCM on other types of software, such as web applications, database systems, or compilers, remains unexplored. While selecting two distinct, large-scale, real-world libraries demonstrates robustness beyond a single domain, generalization to all software domains requires further study.

Architecture Scope. We focus exclusively on migrating from ARM NEON to the RISC-V Vector Extension. FSCM may not generalize to migrations between fundamentally different paradigms, such as CPU-to-GPU or fixed-width SIMD to scalable vector architectures beyond RISC-V. While ARM-to-RISC-V is a commercially and academically significant migration path, and our use of two different RISC-V targets (SpacemiT X60 and T-Head C908) demonstrates generalizability within the RISC-V ecosystem, this does not necessarily extend beyond it. Further studies are needed to evaluate FSCM on other architecture pairs.

Dataset Coverage. Our evaluation dataset consists of only 33 operators from the *Core* and *Imgproc* modules of OpenCV. Additionally, the dataset may not represent the full spectrum of possible vectorized operations. A larger and more diverse dataset, including operators from additional modules or other libraries, would provide a more comprehensive assessment of FSCM's capabilities.

Baseline Comparison. The performance of LLM-based baselines is sensitive to prompt engineering. Although we provided identical context to all methods, different prompt formulations could yield different baseline results.

Performance Measurement. The Speedup Ratio is measured against a scalar baseline, which can be affected by compiler optimizations. We controlled for this by using identical optimization flags, but variations in compiler versions or target-specific tuning could affect the reported speedups.

9 Acknowledgments

We thank all reviewers for their valuable feedback. This work is partially supported by the NSF of China (under Grant 92364202), and Major Program of ISCAS (Grant No. ISCAS-ZD-202402).

References

Anthropic. 2024. Claude 4: Technical report. Technical report, Anthropic. Accessed: 2024-03-20.

Seth Aycock, David Stap, Di Wu, Christof Monz, and Khalil Sima'an. 2025. Can llms really learn to translate a low-resource language from one grammar book? In *ICLR*.

Brenda S Baker. 1996. Parameterized pattern matching: Algorithms and applications. *Journal of computer and system sciences*, 52(1):28–42.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/xt 0.17. a language and toolset for program transformation. *Science of computer programming*, 72(1-2):52–70.

James R Cordy. 2006. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38.

Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards translating real-world code with llms: A study of translating to rust. *arXiv preprint arXiv:2405.11514*.

Stuart I Feldman. 1990. A fortran to c converter. In *ACM SIGPLAN Fortran Forum*, volume 9, pages 21–22. ACM New York, NY, USA.

GitHub. 2025. Github copilot 2025: Advanced ai pair programmer. Technical report, GitHub. Projected future version based on roadmap analysis.

howjmay. 2023. neon2rvv. https://github.com/howjmay/neon2rvv. GitHub repository, accessed 2026-01-06.

Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. 2009. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177. IEEE.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, and 1 others. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, and 1 others. 2025. Large language model agent: A survey on methodology, applications and challenges. *arXiv preprint arXiv:2503.21460*.

OpenAI. 2025. Codex 2025: Technical report and capabilities. Technical report, OpenAI. Hypothetical future version based on current development trends.

Jialing Pan, Adrien Sadé, Jin Kim, Eric Soriano, Guillem Sole, and Sylvain Flamant. 2023. Stelocoder: a decoder-only llm for multi-language to python code translation. *arXiv preprint arXiv:2310.15539*.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 6(3):1.

Ranjan Sapkota, Konstantinos I Roumeliotis, and Manoj Karkee. 2025. Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai. *arXiv preprint arXiv:2505.19443*.

Bo Wang, Tianyu Li, Ruishi Li, Umang Mathur, and Prateek Saxena. 2025. Program skeletons for automated program translation. *Proceedings of the ACM on Programming Languages*, 9(PLDI):920–944.

xAI. 2025. Grok 4. [Large language model].

Qinwei Xu, Ruipeng Zhang, Yi-Yan Wu, Ya Zhang, Ning Liu, and Yanfeng Wang. 2023. Simde: A simple domain expansion approach for single-source domain generalization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4798–4808.

Chen Zhang, Jiuheng Lin, Xiao Liu, Zekai Zhang, and Yansong Feng. 2025a. Read it in two steps: Translating extremely low-resource languages with code-augmented grammar books. *arXiv preprint arXiv:2506.01796*.

Xing Zhang, Jiaheng Wen, Fangkai Yang, Pu Zhao, Yu Kang, Junhao Wang, Maoquan Wang, Yufan Huang, Elsie Nallipogu, Qingwei Lin, and 1 others. 2025b. Skeleton-guided-translation: A benchmarking framework for code repository translation with fine-grained quality evaluation. *arXiv preprint arXiv:2501.16050*.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*.

A Appendix

A.1 Code Refactoring Details.

Global Parsing.

Our approach implements a set of C++ syntax parsing tools that perform hierarchical parsing of the source repository. The parser traverses the directory recursively from top to bottom, capturing entities matching syntax patterns and their interconnections. In this process, the parser skips built artifacts and version control files, preliminarily filtering out non-functional entities. The constructed graph is denoted by $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{T}, \mathcal{R})$ where $\mathcal{G}$ is a directed graph, $\mathcal{V}$ is the set of nodes and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges. For each node $v \in \mathcal{V}$, the mapping $\tau(v) : \mathcal{V} \to \mathcal{T}$ maps the node to a type $t \in \mathcal{T}$ where $\mathcal{T} = \{directory, file, class, struct, function\}$. And for each edge $e \in \mathcal{E}$, the mapping $\rho(e) : \mathcal{E} \to \mathcal{R}$ maps the edge to a relationship $r \in \mathcal{R}$ where $\mathcal{R} = \{contains, invokes, inherits, imports\}$.

Due to the complexity of C++ language, even existing static analysis tools may not be able to accurately capture function invocation patterns. For example, Macro definitions introduce significant complexities in C/C++ parsing. High-performance libraries often contain numerous macros expanded by preprocessors before compilation. Consequently, most static analysis tools (including LLVM IR) cannot directly access original macro definitions when constructing AST and dependency graphs. We handle macros on par with other C++ features in our tools.

Code Refactoring.

Algorithm 1 describes the pruning process.

To determine whether a node can be soundly merged into another, its structural importance is assessed. To achieve this, a metric is proposed to evaluate node importance, taking into account the node's type, out-degree, inheritance relationships, and import dependencies. Nodes assigned

Algorithm 1 Iterative Graph Pruning for Core Function Mining

Initialize $G_{pruned} \leftarrow G_{original}$, $iteration \leftarrow 0$
while $iteration < M$ **do**
  Identify candidate nodes $nodesToMerge$:
  for each node n in G_{pruned} **do**
    // Check if node n is important by its in- and out- degrees
    if n has 0 incoming `INVOKES` **and** `canMerge`$(n, parent)$ **then**
      // Merge with parent directly if it is not invoked
      Merge $(n, parent)$ to $nodesToMerge$
    else if n has 1 incoming `INVOKES` **and** `canMerge`$(n, invokeParent)$ **then**
      // Merge with the invoker if it is invoked by only one node
      Merge $(n, invokeParent)$ to $nodesToMerge$
    end if
  end for
  if $nodesToMerge = \emptyset$ **then**
    // No more nodes to merge, terminate the loop
    break
  end if
  Merge all $(child, parent)$ in $nodesToMerge$
  $iteration \leftarrow iteration + 1$
end while
return G_{pruned}

high importance are preserved during the merging process.

1. **Type-based constraints**: To preserve the fundamental architecture of the code and avoid excessive merging, nodes of file or directory types are assigned high importance.

2. **Out-degree thresholds**: To preserve functionally important nodes, the nodes with out-degrees exceeding a threshold are assigned with high importance.

3. **Inheritance preservation**: To preserve the inheritance structure, the nodes with inheritance relationships are assigned with high importance.

4. **Import preservation**: To preserve modularization, the nodes that are imported by multiple nodes are assigned with high importance.

In the merging process, to ensure the traceability of the merged nodes, all the information of the merged nodes, such as code blocks, edges, and metadata, are preserved in the parent node.

Through the prune-and-merge process, the dependency graph of a large and complex codebase is well-reconstructed, achieving a 30% reduction in node size while preserving all critical architectural relationships, effectively highlighting the core functionality that are most relevant for cross-architecture migration.

A.2 OpenCV

OpenCV[3] (Open Source Computer Vision Library) is the most widely used library for computer vision and image processing worldwide, offering functionalities such as image reading, object recognition, and 3D reconstruction. It is extensively applied in domains including robot vision, industrial inspection, and autonomous driving. With a codebase exceeding 5 million lines and over 2,500 commonly used algorithms, it is organized into multiple modular subsystems, such as core and imgproc. Migration of ImgProc is more difficult yet critical due to complex functions, scattered interdependent SIMD code, and strong performance sensitivity.

We scrutinized the official OpenCV source test cases to identify all cases relevant to the target operators. Specifically, we extract implementations optimized for the ARM NEON architecture and apply a two-stage filtering process: (1) remove legacy files that are no longer utilized in the current OpenCV build, and (2) exclude operators lacking

[3]https://github.com/opencv/opencv

corresponding test cases in the official OpenCV test suite. After filtering, we obtain a dataset of 33 operators in total, including 17 from *Core* and 16 from *Imgproc*. Additionally, we validated outputs to ensure the test cases indeed trigger the intended operator execution. This selection strikes a balance between coverage and feasibility, ensuring that the chosen operators are both widely used and testable.

A.3 Complete Evaluation Results

Table 5 provides additional information for Table 2.

A.4 Result of NCNN

To demonstrate the generalizability of our approach, we further conducted comparative experiments on NCNN, a widely used high-performance library for edge deployment. Due to space limitations, we report only representative results on core operators and commonly used tensor dimensions:

Bias operator:

- Tensor shape (7, 8, 9, 12): Our FSCM achieved a 1.21× speedup over the scalar baseline, surpassing human-expert implementations by 20.11%.
- Tensor shape (3, 5, 1, 13): Our FSCM achieved a 2.1× speedup over the scalar baseline, surpassing human-expert implementations by 62.17%.

Selu operator:

- Tensor shape (127, 1, 1, 1): Our FSCM achieved a 1.3× speedup over the scalar baseline, outperforming human-expert implementations by 10.08%.
- Tensor shape (17, 12, 1, 1): Our FSCM achieved a 1.23× speedup over the scalar baseline, outperforming human-expert implementations by 3.83%.
- Tensor shape (3, 4, 5, 13): Our FSCM achieved a 1.04× speedup over the scalar baseline, outperforming human-expert implementations by 3.46%.

A.5 Evaluation Metrics

To rigorously evaluate effectiveness in cross-architecture code migration, we adopt a multi-dimensional evaluation methodology. Specifically, we consider three key metrics:

Compilation Status (CS)

For each migrated function i we define a binary compilation indicator

$$\mathrm{CS}_i = \begin{cases} 1, & \text{if function } i \text{ compiles successfully,} \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

where N is the total number of migrated functions. Note that CS is a gate: only functions with $\mathrm{CS}_i = 1$ are considered for subsequent functional and performance evaluation.

Test Success Rate (TS)

This metric evaluates the semantic correctness of the migrated code that passes the official OpenCV test suite. Formally:

$$\mathrm{TS} = \frac{N_{\text{passed}}}{N_{\text{tests}}} \tag{2}$$

where N_{passed} is the number of passed test cases and N_{tests} is the total number of test cases.

Speedup Ratio (SR)

This metric measures the relative execution speed of the migrated, vectorized implementation compared with a scalar baseline. We compute SR as:

$$\mathrm{SR} = \frac{T_{\text{scalar}}}{T_{\text{vector}}} \tag{3}$$

where T_{scalar} is the execution time of the scalar implementation and T_{vector} is that of the migrated vectorized implementation.

A.6 RVV Knowledge

With a functionally correct and algorithmically refined baseline, the framework then injects optimizations tailored specifically for the RISC-V Vector Extension. This is the most critical phase for performance enhancement and includes several advanced techniques:

- **Register Blocking:** To maximize data reuse and minimize costly memory-to-register traffic, we group operations to work on data chunks that fit within RVV's large vector register file.
- **Variable-Length Vectorization:** A key advantage of RVV is its vector-length agnostic (VLA) design. Our framework generates code that adapts automatically to the hardware's specific vector length (`VLEN`).
- **Intrinsic Selection:** The framework intelligently selects the most efficient RVV intrinsics for a given operation.

Table 5: The complete performance comparison across different RISC-V and ARM platforms, LLMs, operators, and matrix sizes. Speedup over scalar implementations (×). "*" indicates results with Architecture-specific Optimization. "failed-c" indicates compilation failure; "failed-t" indicates functional test failure. Gray background highlights the best performance.

Target Platform	Method	Add		Sub		Max		Mul	
		1280x7200 8SC1	1280x720 8UC1	640x480 8UC4	1280x720 8UC1	640x480 8UC1	640x480 8UC3	1280x720 8UC1	1280x720 8UC4
X60 (RISC-V)	Compiler-O3	7.92×	13.10×	11.80×	11.86×	10.01×	10.12×	4.64×	4.62×
	Codex	failed-t	failed-t	failed-c	failed-c	85.12×	76.28×	failed-c	failed-c
	Copilot	failed-t	failed-t	100.45×	102.15×	90.80×	40.38×	61.25×	58.41×
	Human Expert	63.87×	50.90×	121.00×	124.50×	85.12×	74.89×	59.69×	55.14×
	Ours								
	+Grok 4	80.62×	52.93×	96.80×	102.15×	71.68×	73.55×	21.06×	20.56×
	+Grok 4 *	80.62×	88.22×	102.38×	102.15×	85.12×	76.28×	21.50×	20.56×
	+Claude Sonnet 4	65.57×	69.65×	133.10×	132.80×	90.80×	76.28×	78.10×	82.65×
	+Claude Sonnet 4 *	106.91×	79.40×	133.10×	132.80×	90.80×	79.21×	88.42×	85.68×
	relative improvement	**(↑43.04×)**	**(↑28.50×)**	**(↑12.10×)**	**(↑8.30×)**	**(↑5.68×)**	**(↑4.32×)**	**(↑28.73×)**	**(↑30.54×)**

Target Platform	Method	AddWeighted		DotProduct		Cmp		Absdiff	
		1920x1080 8UC4	640x480 8UC4	512 8SC1	1024 8SC1	640x480 8UC1	640x480 8UC4	640x480 8UC1	640x480 8UC4
X60 (RISC-V)	Compiler-O3	4.92×	4.87×	5.85×	5.78×	9.50×	9.71×	6.88×	6.90×
	Codex	failed-c	failed-c	41.36×	42.81×	failed-c	failed-c	failed-t	failed-t
	Copilot	62.82×	61.97×	failed-t	failed-t	55.81×	51.42×	81.25×	75.29×
	Human Expert	58.80×	58.66×	48.25×	48.17×	55.81×	51.42×	81.25×	76.40×
	Ours								
	+Grok 4	64.58×	64.48×	44.54×	46.24×	49.61×	49.34×	48.15×	65.76×
	+Grok 4 *	65.55×	64.48×	48.25×	47.18×	59.53×	52.91×	81.25×	72.15×
	+Claude Sonnet 4	69.87×	71.57×	52.64×	56.39×	55.81×	55.32×	76.47×	77.54×
	+Claude Sonnet 4 *	72.66×	73.41×	52.64×	56.39×	55.81×	55.32×	81.25×	77.54×
	relative improvement	**(↑13.86×)**	**(↑14.75×)**	**(↑4.39×)**	**(↑8.22×)**	**0**	**(↑3.9×)**	**0**	**(↑1.14×)**

Target Platform	Method	Sub		AddWeighted		Mul		Resize	
		1920x1080 8SC1	1280x720 8SC1	640x480 8UC4	640x480 8UC1	640x480 32FC1	1920x1080 32FC1	640x480 16UC1	640x480 8UC4
C908 (RISC-V)	Compiler-O3	7.73×	7.78×	4.16×	4.18×	8.98×	9.10×	6.40×	5.78×
	Codex	failed-c	failed-c	failed-c	failed-c	failed-c	failed-c	failed-t	failed-t
	Copilot	92.96×	91.93×	26.19×	25.58×	16.35×	16.63×	failed-t	failed-t
	Human Expert	116.92×	90.29×	26.29×	25.40×	16.35×	16.65×	3.84×	2.94×
	Ours								
	+Grok 4	92.96×	91.93×	29.73×	29.29×	28.28×	28.66×	6.19×	6.25×
	+Grok 4 *	92.96×	91.93×	29.73×	29.53×	28.28×	28.66×	6.19×	6.25×
	+Claude Sonnet 4	119.38×	101.12×	31.26×	31.58×	26.83×	27.18×	0.91	0.61
	+Claude Sonnet 4 *	119.38×	114.91×	31.26×	31.58×	26.83×	27.18×	0.91×	0.61×
	relative improvement	**(↑2.46×)**	**(↑24.62×)**	**(↑4.97×)**	**(↑6.18×)**	**(↑10.48×)**	**(↑12.01×)**	**(↑2.35×)**	**(↑3.31×)**

A.7 An example

Below we summarize the example in Figure 2 using *Gaussian Blur*.

In the original ARM NEON implementation, [6] *Gaussian Blur* comprises coupled functions, "isGBlur3x3", "gBlur3x3", "isGBlur3x3Margin", "gBlur3x3Margin", and their 5×5 variants, interwoven with architecture-specific structs, macros, and intrinsic wrappers. FSCM refactors the code (Section 3.1) to extract architecture-independent invariant semantic anchors by removing these artifacts. The operator is reconstructed around semantic cores:

```
gaussian_blur{
  "name": "gaussian_blur",
  "components": [
    "gaussianBlur3x3",
    "gaussianBlur5x5",
    ...
  ]
}
```

Each component is a node with 4 architecture-agnostic attributes: "name", "description", "function", and "optimization". For example, the "gaussianBlur3x3" node is defined as:

```
{
  "name": "gaussianBlur3x3",
  "description": "Provides 3x3
     Gaussian blur filtering to
     smooth the input image and
     reduce noise.",
  "function": "Implements 3x3
     Gaussian blur with constant
     padding and border
     replication using a [1 2 1]
     kernel. A separable
     convolution is applied:
     vertical weighted summation
     produces intermediate
     results, followed by
     horizontal weighted
     summation, with
     normalization by 16.",
  "optimization": "Convert scalar
      operations into vector
     operations: use vector
     additions to accumulate
     multiple pixels in parallel;
      use vector extract and
     shuffle instructions to
     implement sliding windows
     and fetch neighboring pixels
      in parallel."
}
```

FSCM first synthesizes a scalar implementation from the 'description' and 'function' fields to ensure semantic fidelity. It then performs ISA-specific optimization guided by the 'optimization' field and the target architecture's vectorization strategy. By decoupling functional semantics from architecture-dependent optimization, FSCM mitigates surface-level matching bias.