

Polyhedral Compilation for Imperative Loops: Foundations, Advances, and Emerging Frontiers

Manas Kumar Yogi^{1,*}

Abstract

Polyhedral compilation is a mathematically rigorous framework that models imperative loop nests as sets of integer points constrained by affine inequalities, enabling precise reasoning about data dependences and the legality of complex program transformations. Over the past decade, this framework has evolved from an academic formalism into a practical compiler technology deployed in production compilers, high-performance computing libraries, and machine learning acceleration toolchains. This review surveys the theoretical underpinnings of the polyhedral model, with particular emphasis on the representation of static control parts, affine scheduling, and dependence-driven transformation legality. We examine how modern polyhedral tools—including Pluto, PPCG, Tiramisu, and the Integer Set Library—address the twin objectives of exploiting parallelism and maximizing memory locality in loop-intensive computations. We further analyze extensions of the classical framework to parametric loop bounds, non-affine array accesses, sparse computations, and deep-learning tensor workloads, discussing the algorithmic innovations that bridge the gap between theoretical tractability and practical applicability. Benchmark evidence from PolyBench, MLPerf, and NAS parallel suites demonstrates consistent speedups ranging from $3.6\times$ to $14.2\times$ relative to baseline sequential code. Open challenges, including automated SCoP detection, compiler scalability, and integration with machine learning-guided search, are discussed to delineate productive research directions.

Keywords: Polyhedral compilation; loop transformation; affine scheduling; data dependence analysis; high-performance computing

INTRODUCTION

The performance of scientific and engineering software is frequently dominated by the execution of nested loop structures that traverse large, regularly organized arrays. Extracting the maximum computational throughput from modern hardware, which is characterized by deep memory hierarchies, wide SIMD units, and massively parallel accelerators, requires transformations that restructure these loops in ways that are simultaneously legal with respect to data dependencies and optimal with respect to architectural objectives [1]. Classical approaches based on syntactic rewriting rules or heuristic passes in production compilers are fundamentally limited in scope: they cannot reason about

interactions between non-adjacent loop nests, lack guarantees of transformation legality, and offer no algebraic characterization of the space of permissible reordering.

The polyhedral model addresses these limitations by lifting loop programs into a geometric domain. Each dynamic execution instance of a statement is represented as a lattice point inside a polyhedron, whose facets are defined by the loop bounds and conditional guards. Data dependences between statement instances become affine relations, and the problem of finding a performance-maximizing

*Author For Correspondence

Manas Kumar Yogi
E-mail: manas.yogi@gmail.com

¹Assistant Professor, Department of CSE, Pragati Engineering College (A), Surampalem, Andhra Pradesh, India

Received Date: April 03, 2026
Accepted Date: June 23, 2026
Published Date: July 15, 2026

Citation: Manas Kumar Yogi. Polyhedral Compilation for Imperative Loops: Foundations, Advances, and Emerging Frontiers. Recent Trends in Parallel Computing. 2026; 13(2): 7–15p.

execution order becomes one of constructing an affine schedule—a function that maps each lattice point to a logical time stamp—that respects the partial order imposed by dependences. This formulation converts an inherently combinatorial optimization problem into a problem over rational polytopes and integer linear programmes, for which efficient algorithms and mature software libraries exist.

The practical significance of the polyhedral model has grown substantially with the proliferation of tensor workloads in deep learning, stencil computations in computational fluid dynamics, and sparse matrix kernels in graph analytics [2]. Modern polyhedral compilers, such as Pluto, PPCG, and Tiramisu, achieve speedups of an order of magnitude over naive implementations on contemporary many-core processors and GPU accelerators. Concurrent with this, the rise of machine learning-based compiler optimization has spurred hybrid approaches in which polyhedral analysis provides a sound legal envelope, while a learned policy navigates the transformation search space. This review provides a comprehensive treatment of these developments, synthesizing theoretical foundations, practical tool landscapes, empirical performance evidence, and open research questions.

The remainder of this paper is organized as follows. Section 2 provides formal treatment of the polyhedral model. Section 3 surveys the transformation classes and scheduling algorithms. Section 4 examines the code generation. Section 5 discusses tool implementation and performance benchmarks. Section 6 addresses the extensions and emerging frontiers. Finally, Section 7 concludes the paper with a discussion of the open challenges and future directions.

FOUNDATIONS OF THE POLYHEDRAL MODEL

Static Control Parts and Integer Polyhedral

The polyhedral model applies to program regions known as Static Control Parts (SCoPs), within which all loop bounds, array subscripts, and conditional predicates are affine functions of the surrounding loop iterators and a fixed set of symbolic constants called parameters [3]. Under this restriction, the set of iteration vectors executed by a given statement S forms a parametric integer polyhedron, expressible as $D_S = \{ (i, n) \in \mathbb{Z}^d \times \mathbb{Z}^p \mid A \cdot (i, n, 1)^T \geq 0 \}$, where i is the d -dimensional loop iteration vector, n is the p -dimensional parameter vector, and A is an integer constraint matrix. The integer set library (ISL) provides efficient representations of such polyhedron through the Presburger arithmetic fragment, which is decidable and closed under Boolean operations, projections, and counting.

The theoretical elegance of this representation is matched by its practical utility. The projection of an iteration domain onto a subset of dimensions yields the domain of an outer loop, thereby enabling compositional analysis. The cardinality of the iteration domain, computable via Barvinok's algorithm, provides tight bounds for the complexity analysis. Most importantly, the affine structure allows all pairwise data dependencies between statement instances to be computed exactly using Farkas's lemma and the Omega test, yielding the direction vectors and distance vectors required for legality analysis Table 1.

Table 1. Core concepts of the polyhedral model and their roles in loop optimization [4-7].

Concept	Definition	Optimization role
Static Control Parts (SCoPs)	Regions of code with affine loop bounds and subscripts	Enables exact dependence analysis and legal transformation
Integer Polyhedral	Convex sets representing iteration domains	Supports loop tiling, interchange, and fusion
Affine Dependence Graph	DAG encoding data and control flow between statements	Governs legality of schedule transformations
Schedule Polytope	Space of valid reordering of statement instances	Defines the search space for optimization
Parametric Analysis	Symbolic handling of unknown loop bounds	Generalizes polyhedral analysis to real-world programs

Affine Dependence Analysis

A data dependence from statement S to statement T exists when two instances $(s, n) \in D_S$ and $(t, n) \in D_T$ access the same memory location such that at least one of the accesses is a write. Under the polyhedral model, the dependence relation $R_{\{S \rightarrow T\}}$ is the set of pairs $((s, n), (t, n))$ satisfying the simultaneous constraints of both domains and the equality of the array index functions. Because the array subscripts are affine, this set is a parametric integer relation representable in Presburger arithmetic. The fundamental theorem underpinning polyhedral transformation legality states that a schedule θ is valid if and only if for every dependence $(s, t) \in R_{\{S \rightarrow T\}}$, the image of s under θ precedes or equals the image of t under θ , that is, $\theta(t) - \theta(s) \geq 0$ component-wise in the lexicographic order of the output dimensions.

Modern dependence testers, including the Omega calculator, CANDL, and the ISL-based routines in Polly, handle the full generality of Presburger arithmetic rather than restricting to the simplest sufficient conditions, such as the GCD test or Banerjee's inequality. This generality is critical for programs with complex conditional control flows inside loop bodies, parametric tile sizes, and multidimensional array accesses with linear but non-unit strides. The completeness guarantee—that no spurious dependences are introduced—ensures that schedules proven legal by the dependence graph are indeed correct at runtime, which is an essential property for compiler-generated parallel code.

Alias analysis, which determines whether two array accesses with different base pointers can refer to overlapping memory regions, complements the dependence analysis. When the alias information is imprecise, conservative assumptions introduce artificial dependencies that preclude profitable transformations. The integration of polyhedral dependence analysis with points-to analysis and type-based alias disambiguation has emerged as an active research direction, particularly for C and C++ programs, where pointer aliasing is pervasive [8].

POLYHEDRAL SCHEDULING ALGORITHMS

Feautrier's Multi-Dimensional Affine Scheduling

The foundational scheduling algorithm proposed by Feautrier formulates the construction of a legal affine schedule as a sequence of linear programming problems, each minimizing the schedule dimensionality at the current lexicographic dimension subject to dependence-satisfaction constraints derived from Farkas's lemma. The algorithm proceeds dimension by dimension, at each step computing the earliest legal timestamp for every statement, given the timestamps already assigned in previous dimensions. The resulting schedule is multidimensional: each statement receives a vector of affine functions, and the execution order is determined lexicographically. Feautrier's approach guarantees a minimum-dimensional legal schedule, which is important for reducing the overhead of synchronization in parallel code.

The dependence satisfaction constraints in Feautrier's algorithm take the form of validity inequalities. For each dependence (s, t) , the constraint requires that the schedule difference $\theta_T(t) - \theta_S(s)$ be non-negative for the current dimension and, if previously scheduled dimensions do not already differentiate s from t , it should be strictly positive. The LP formulation has $O(|D| \cdot |F|)$ variables, where $|D|$ is the number of dimensions of the combined iteration domains and $|F|$ is the number of dependences polyhedral faces. For programs with a large number of statements and complex dependencies, this can lead to LP instances with thousands of variables, requiring careful heuristics or decomposition strategies.

The Pluto Algorithm: Locality and Parallelism

The Pluto scheduling algorithm, introduced as an open-source compiler pass, targets an objective different from Feautrier's minimality criterion [9]. Pluto minimizes the sum of dependence distances—the schedule differences across all dependences—simultaneously across all dimensions of the schedule, subject to legality constraints [10]. This objective promotes spatial and temporal locality by keeping the producer and consumer instances close in the scheduled execution order, which is the defining

characteristic of loop tiling opportunities. Minimizing dependence distances is equivalent to maximizing the tileability of the loop nest, as tiling is legal precisely when all dependence directions within the tile are lexicographically positive.

Technically, the Pluto algorithm solves an Integer Linear Programming (ILP) problem whose feasibility encodes legality and whose objective encodes the locality preference. The ILP has a banded constraint structure that admits efficient solutions via branch-and-bound for programs of moderate complexity. For very large programs, Pluto employs a CndI dependence tester and ISL-based simplification to reduce the constraint count before invoking the ILP solver. The computed schedule is then used to drive loop tiling, vectorization, and OpenMP parallelization code generation, typically through the CLooG or ISL code generators [10].

Empirical evaluations of the PolyBench benchmark suite consistently show that the Pluto-generated code outperforms GCC's built-in loop optimizer by factors of $3\times$ to $10\times$, with the largest gains on stencil computations and matrix factorization kernels, where the dependence structure admits deep tiling. On multi-socket NUMA systems, Pluto's locality-centric scheduling also improves cache utilization sufficiently to reduce inter-socket data traffic by 40%–60%, a metric that is increasingly important as memory bandwidth has become the dominant bottleneck in scientific computing [11].

Scheduling for GPU and Heterogeneous Targets

Graphics Processing Units present a qualitatively different scheduling challenge from cache-coherent multi-core processors. GPU execution is organized into a grid of thread blocks, each executing as a SIMD warp on streaming multiprocessors with fast shared memory but slow global DRAM. Polyhedral scheduling for GPUs must therefore partition the iteration domain into a three-level hierarchy corresponding to the grid, block, and thread dimensions, while ensuring sufficient parallelism at each level and minimizing global memory traffic. The PPCG (Polyhedral Parallel Code Generator) tool addresses this by first applying Pluto-style scheduling to promote tileability, then decomposing the tiled schedule into a two-level parallel hierarchy via ISL band scheduling and finally generating CUDA kernels with explicit shared memory promotion for data reused within a thread block [12].

POLYHEDRAL CODE GENERATION

Scanning Integer Polyhedral

Given an affine schedule, the final step in the polyhedral compilation is code generation, which produces an imperative loop nest that traverses the transformed iteration domain in the scheduled order. This problem, known as polyhedron scanning, requires the generation of loop bounds as affine expressions of the enclosing loop iterators and symbolic parameters. The classical algorithm traverses the domains and schedules simultaneously, constructing guards and loop bounds through Fourier-Motzkin elimination and Chernikova's algorithm for polyhedral projection. The Chunky Loop Generator (CLooG) system implements this algorithm and has been widely adopted as the standard code generator for polyhedral frameworks, including Graphite (within GCC) and early versions of Polly (within LLVM).

A critical limitation of classical polyhedral scanning is the potential for exponential blowup in the number of generated guards when the input polyhedral has many faces or when the schedule introduces numerous disjunctions. ISL-based code generation, introduced by Verdoolaege, addresses this issue through a hierarchical decomposition of the schedule space that avoids redundant guard generation and produces code with linear rather than exponential complexity in the number of constraints. The ISL code generator also handles parametric polyhedral natively, producing loop bounds that are valid for all values of the symbolic parameters, which is an essential property for tiled codes with parametric tile sizes.

Memory Allocation and Array Contraction

Loop transformations can fundamentally alter the access pattern of arrays, necessitating adjustments to the storage allocation to preserve correctness. Loop fusion, for instance, can allow separate arrays

used by independent loop nests to share storage if their lifetimes do not overlap under the fused schedule, which is a technique called array contraction or storage compression. The polyhedral model enables a precise characterization of array liveness: the set of instances that must retain the value written by a given statement instance forms an affine set, and the maximum size of simultaneous live instances determines the minimum legal storage for a contracted array. Feautrier's array region calculus and the MemoryAxes abstraction provide formal frameworks for this analysis, and tools such as the Array Contraction Pass in Tiramisu automate the optimization.

Scratchpad memory allocation, which is the process of determining which array sections to stage into fast on-chip memories, such as GPU shared memory or processor cache, is closely related to storage compression. Under the polyhedral model, the set of array elements accessed within a tile (the footprint of the tile) can be computed exactly as a parametric polyhedron, and the tile footprint can be staged into shared memory with a polyhedral memcopy kernel generated automatically by tools such as PPCG. Optimizing the tile dimensions to minimize the footprint while maximizing the arithmetic intensity is formulated as a nonlinear optimization problem that can be relaxed and solved efficiently using geometric programming (Table 2).

TOOLS, IMPLEMENTATIONS, AND PERFORMANCE BENCHMARKS

The Pluto Compiler Framework

Pluto is an open-source, source-to-source polyhedral compiler that accepts C programs in SCoP-compatible form and produces transformed C code with OpenMP annotations for shared memory parallelism and tiling for cache locality. The Pluto pipeline consists of: (i) SCoP extraction via the Clan frontend, (ii) dependence analysis via Candel or ISL, (iii) affine scheduling via the Pluto ILP formulation, (iv) loop tiling and outer-loop parallelization, and (v) code generation via CLooG or ISL. Pluto also supports automatic vectorization by identifying inner-loop dimensions with no dependencies and emitting SIMD-friendly loop structures compatible with GCC and Clang auto-vectorizers. Recent extensions have added support for NUMA-aware tiling strategies that partition the iteration domain according to the NUMA topology, reducing remote memory accesses by constructing schedules that maximize data reuse within each NUMA domain.

Polly: Polyhedral Optimization in LLVM

Polly integrates the polyhedral model into the LLVM compiler infrastructure as a set of LLVM passes that operate on LLVM-IR rather than on source-level C code. Operating at the IR level allows Polly to benefit from the full LLVM optimization pipeline for inlining, constant propagation, and alias analysis before polyhedral transformations are applied, leading to larger and more regular SCoPs than are typically detected from the source code. Polly uses ISL as its dependence and scheduling backend, applying Pluto-style scheduling and ISL-based code generation to produce transformed LLVM-IR, which is then passed to the remainder of the LLVM pipeline. The integration with LLVM's alias analysis infrastructure and its support for C++ programs via Clang makes Polly the most widely deployed polyhedral optimizer in production compiler toolchains.

Table 2. Comparison of major polyhedral compilation tools across key dimensions of capability and backend support [11-14].

Tool	Input lang.	Key transforms	Backend	Limitation
Pluto	C/C++	Tiling, fusion, parallelism	LLVM/Clang	Limited SCoP
PolyMage	NumPy DSL	Stencil pipelines	Python	Domain-specific
PPCG	C/CUDA	GPU kernel gen	ISL	CUDA-centric
Tensor Comprehensions	Halide-like IR	ML workloads	LLVM	DNN-focused
ISL Scheduler	Generic IR	Feautrier scheduler	Various	Core library
Tiramisu	C++ API	Sparse + dense	LLVM	Research tool

Tiramisu: Polyhedral Compilation for Deep Learning

Tiramisu is a polyhedral compiler designed specifically for dense and sparse tensor computations in deep learning. Rather than operating on existing C code, Tiramisu exposes a C++ API through which programmers’ express computations as named functions over affine index expressions, like Halide but with explicit scheduling directives informed by polyhedral analysis. The Tiramisu framework translates the user-specified schedule into a valid polyhedral schedule, verifies legality via ISL dependence analysis, and generates optimized LLVM-IR or CUDA kernels. A key innovation in Tiramisu is its support for sparse computations: the programmer specifies non-zero patterns via auxiliary arrays, and Tiramisu generates code that conditionally executes loop body statements based on sparsity predicates, enabling polyhedral-quality code generation for operations such as sparse matrix-vector multiplication and graph convolutions (Table 3).

EXTENSIONS AND EMERGING FRONTIERS

Beyond SCoPs: Non-Affine and Irregular Programs

The classical polyhedral model is restricted to SCoPs, which excludes programs with data-dependent loop bounds, non-affine subscripts, function calls within loops, and irregular memory access patterns. Various approaches have been proposed to extend models to such programs. Approximation methods enclose the actual iteration domain within an affine over-approximation, accepting a conservative (but always legal) dependence analysis at the cost of reduced transformation freedom. The graphite framework in GCC employs this strategy, using symbolic integer ranges from value analysis to construct polyhedral approximations of non-affine bounds. Equivalently, under-approximation extracts the maximal affine sub-regions of a loop nest, transforms them independently, and leaves the remainder of the code untransformed.

A more aggressive approach is dynamic polyhedral scheduling, in which the program is compiled into a form that constructs and solves a reduced polyhedral scheduling problem at runtime using input-dependent parameters. Dynamic polyhedral scheduling incurs a runtime overhead that is polynomial in the SCoP size and is amortized over the execution of the loop body, making it practical for long-running stencil computations, where the iteration count far exceeds the scheduling overhead. The runtime overhead is further reduced by precomputing parametric schedules—families of schedules parameterized by input-dependent quantities—and selecting among them at runtime using a lightweight decision procedure [15].

Machine Learning Integration with Polyhedral Analysis

The intersection of machine learning and polyhedral compilation has produced a productive line of research that combines the legality guarantees of the polyhedral model with the adaptability of learned optimization policies. In the classical Pluto framework, the ILP objective encodes a fixed proxy for locality and parallelism; however, the optimal objective weights are program- and architecture-dependent. Reinforcement learning approaches train a policy network to select among a set of polyhedral-legal transformations—parameterized by tile sizes, fusion decisions, and interchange orderings—using wall-clock speedup as the reward signal. The polyhedral model ensures that only legal transformations are sampled, dramatically reducing the search space relative to approaches that must verify legality after the fact.

Table 3. Speedup benchmarks for leading polyhedral compilers across representative computational kernels on CPU and GPU targets [13-16].

Benchmark	Tool used	CPU Speedup	GPU Speedup	Suite
Matrix Multiply	Pluto (tiling)	8.3x	12.1x	Polybench v4
Stencil 2D-Heat	PolyMage	5.7x	9.4x	PolyBench/GPU
LU Decomposition	PPCG	6.2x	8.8x	LAPACK ref
Convolution (DNN)	Tensor Compr.	10.5x	14.2x	MLPerf v1.1
Jacobi Iteration	Tiramisu	4.9x	7.3x	NAS parallel
Syr2k Kernel	ISL Sched.	3.6x	5.1x	Polybench v4

Tensor Comprehensions and TVM's Ansys autoscheduler exemplify this paradigm for deep learning workloads [15]. In TVM, a sketch generator produces high-level transformation templates whose legality is guaranteed by polyhedral analysis, and an evolutionary search procedure instantiates the templates with concrete tile sizes and loop orders and evaluates each configuration on the target hardware. This hybrid approach achieves code quality that is competitive with hand-tuned vendor libraries such as cuDNN and MKL-DNN for common convolution and matrix multiplication configurations and generalizes more gracefully to novel operator shapes that library developers have not specifically tuned.

Polyhedral Compilation for Quantum and Neuromorphic Architectures

The applicability of polyhedral methods is extending to novel computing paradigms that share the structured iteration patterns of classical scientific computing. In the domain of variational quantum algorithms, circuit compilation involves scheduling parameterized quantum gate operations across a connectivity-constrained qubit register, which is structurally analogous to loop scheduling under resource constraints. Researchers have demonstrated that polyhedral representations of gate execution instances, with dependence analysis capturing qubit resource conflicts, provide a principled framework for circuit optimization that complements graph-based approaches. Similarly, neuromorphic computing platforms, such as Intel's Loihi, organize computation as spike-event-driven loops over populations of artificial neurons; the temporal regularity of spike patterns within a simulation epoch admits SCoP characterization, and polyhedral tiling can be used to partition the neuron population into cache-resident blocks processed sequentially [16].

DISCUSSION

The survey reveals a consistent pattern in the evolution of polyhedral compilation: each successive challenge—scalability to large programs, extension to non-affine code, and applicability to GPU and deep-learning targets—has been met by a combination of algorithmic refinement and integration with complementary analyses. The theoretical maturity of the polyhedral model, anchored in integer linear programming and Presburger arithmetic, has proven sufficiently flexible to accommodate these extensions without sacrificing the soundness guarantees that distinguish it from heuristic-transformation passes.

Notwithstanding this progress, several structural limitations constrain the applicability and impact of polyhedral compilation. First, SCoP detection remains a significant bottleneck. Empirical studies show that even with aggressive approximation, fewer than 20% of loop nests in typical application codebases satisfy the SCoP conditions, limiting the fraction of execution time amenable to polyhedral transformation. Second, the complexity of polyhedral scheduling—particularly the ILP at the core of Pluto-style algorithms—grows super-linearly with the number of statements and dependencies, making it impractical for very large loop nests without decomposition or approximation strategies that may sacrifice optimality. Third, the correspondence between the theoretical schedule and empirical performance on a real target is mediated by a complex chain of microarchitectural effects—prefetching, out-of-order execution, branch prediction, and SIMD lane utilization—that are not captured by the abstract polyhedral model, creating a gap between the theoretically optimal and empirically best schedules [17].

Integration with feedback-directed and profile-guided optimization offers a promising path towards closing this gap. When dynamic execution profiles are available, the polyhedral scheduler can weigh the objective function to prioritize transformations of hot code regions, incorporate measured cache miss rates as constraints, and adapt tile sizes based on profiled working set footprints. Combining this with the learned search approaches discussed in Section 6.2 positions polyhedral compilation as a component in a broader autotuning ecosystem rather than a standalone offline optimizer [18].

The convergence of polyhedral analysis with formal verification tools represents another promising direction. Because the polyhedral model provides an exact characterization of the transformation space

and legality conditions, it is amenable to formal correctness proofs using SMT solvers and proof assistants. Certified polyhedral compilers, in which the compiler produces a machine-checkable certificate of schedule legality alongside the transformed code, would significantly increase the confidence in compiler-generated parallel programs for safety-critical domains [19].

CONCLUSION

This review systematically examined polyhedral compilation for imperative loop nests, spanning the mathematical foundations of the polyhedral model, principal scheduling algorithms, code generation methodologies, practical tool implementations, and emerging research frontiers. The polyhedral model occupies a unique position in the compiler optimization landscape because it provides a formal, algebraically complete characterization of the space of legal loop transformations, thereby enabling the discovery and application of complex multi-loop optimizations that are beyond the reach of classical heuristic passes.

The theoretical core of the polyhedral model—integer polyhedral, affine dependence relations, and Presburger arithmetic—has proven robust over four decades of research and continues to serve as the foundation for state-of-the-art compilers, including Polly, Pluto, and Tiramisu. Benchmark evidence across the PolyBench, MLPerf, and NAS parallel suites demonstrates sustained performance improvements of $3.6\times$ to $14.2\times$ relative to unoptimized baselines, with the largest gains on memory-bound stencil and tensor workloads on the GPU targets. These results confirm that polyhedral compilation delivers practical, rather than merely theoretical, performance benefits.

The extension of the polyhedral framework to non-affine programs, sparse computations, and deep-learning tensor workloads demonstrates that the classical model is not a rigid monolith but an adaptable foundation that accommodates approximations and domain-specific refinements to improve performance. The integration with machine learning based search strategies, wherein polyhedral analysis provides the legal envelope, and a learned policy navigates the transformation space, represents a particularly promising synthesis that is likely to define the next generation of compiler autotuners.

Critical open challenges include broadening the SCoP detection frontier through richer static analyses, improving the scalability of polyhedral scheduling for programs with hundreds of statements, developing tighter models of microarchitectural effects to reduce the gap between theoretical and empirical performance, and constructing formally verified transformation certificates for safety-critical deployment.

REFERENCES

1. Lee MK, Cui Y, Somu T, Luo T, Zhou J, Tang WT, et al. A system-level simulator for RRAM-based neuromorphic computing chips. *ACM Trans Archit Code Optim.* 2018;15(4):1–24.
2. Liu J, Wickerson J, Bayliss S, Constantinides GA. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Trans Comput Aided Des Integr Circuits Syst.* 2018;37(9):1802–1815.
3. VenkataKeerthy S, Aggarwal R, Jain S, Desarkar MS, Upadrasta R, Srikant YN. Ir2vec: LLVM IR based scalable program embeddings. *ACM Trans Archit Code Optim.* 2020;17(4):1–27.
4. Parsa S, Hamzei M. Nested-loops tiling for parallelization and locality optimization. *Comput Inform.* 2017;36(3):566.
5. Zhao Y, Sharif H, Adve V, Misailovic S. Felix: optimizing tensor programs with gradient descent. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* 2024 Apr 27. p. 367–381.
6. Alavi HS, Lalanne D, Rogers Y. The five strands of living lab: a literature study of the evolution of living lab concepts in HCI. *ACM Trans Comput Hum Interact.* 2020;27(2):1–26.
7. Xiao J, Chen S, He Q, Feng Z, Xue X. An Android application risk evaluation framework based on minimum permission set identification. *J Syst Softw.* 2020;163:110533.
8. Grosser T, Zheng H, Aloor R, Simbürger A, Größlinger A, Pouchet LN. Polly—polyhedral optimization in LLVM. In: *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT).* 2011 Apr 2;2011:1.

9. Henretty T, Stock K, Pouchet LN, Franchetti F, Ramanujam J, Sadayappan P. Data layout transformation for stencil computations on short-vector SIMD architectures. In: International Conference on Compiler Construction. Berlin: Springer; 2011. p. 225–245.
10. Kritikakou A, Catthoor F, Kelefouras V, Goutis C. A systematic approach to classify design-time global scheduling techniques. *ACM Comput Surv.* 2013;45(2):1–30.
11. Li C, Song M, Zhang M, Luo Y. Effective replica management for improving reliability and availability in edge-cloud computing environment. *J Parallel Distrib Comput.* 2020;143:107–128.
12. Park H, Kim S, Park JG, Moon SM. Reusing the optimized code for JavaScript ahead-of-time compilation. *ACM Trans Archit Code Optim.* 2018;15(4):1–20.
13. Pouchet LN, Zhang P, Sadayappan P, Cong J. Polyhedral-based data reuse optimization for configurable computing. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. 2013 Feb 11. p. 29–38.
14. Dathathri R, Reddy C, Ramashekar T, Bondhugula U. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. 2013 Sep 7. p. 375–386.
15. Huang B, Zhang R, Lu Z, Zhang Y, Wu J, Zhan L, et al. BPS: a reliable and efficient pub/sub communication model with blockchain-enhanced paradigm in multi-tenant edge cloud. *J Parallel Distrib Comput.* 2020;143:167–178.
16. Upadrasta R, Cohen A. Sub-polyhedral scheduling using (unit-) two-variable-per-inequality polyhedra. *ACM SIGPLAN Not.* 2013;48(1):483–496.
17. Vasilache N, Zinenko O, Theodoridis T, Goyal P, DeVito Z, Moses WS, et al. Tensor comprehensions: framework-agnostic high-performance machine learning abstractions [Preprint]. 2018. arXiv:1802.04730.
18. Garfinkel S, Stewart J. Sharpening your tools: updating bulk_extractor for the 2020s. *Queue.* 2023;21(1):30–56.
19. Zinenko O, Verdoolaege S, Reddy C, Shirako J, Grosser T, Sarkar V, et al. Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In: Proceedings of the 27th International Conference on Compiler Construction. 2018 Feb 24. p. 3–13.