Evaluating Polyhedral Optimizers via the Kiriko Tool

1st Lucas Victor da Silva Costa
Department of Computer Science
UFMG
Belo Horizonte, Brazil
1victor.lucassilva@gmail.com

2nd Michael Canesche
Department of Computer Science
UFMG
michael.canesche@gmail.com

3rd Fernando Magno Quintão Pereira
Department of Computer Science
UFMG
fernando@dcc.ufmg.br

***Abstract*—Polyhedral optimizations play a central role in improving the performance of loop-intensive programs, yet existing tools such as Pluto, LLVM Polly, and the MLIR Affine dialect rely on heterogeneous infrastructures, input formats, and compilation pipelines. This fragmentation hinders reproducible experiments and fair comparisons across polyhedral frameworks. To address these challenges, we introduce `Kiriko`, a unified and automated benchmarking infrastructure designed to systematically evaluate polyhedral optimizers under consistent experimental conditions. `Kiriko` integrates multiple toolchains into a single workflow, providing end-to-end support for optimization, lowering, compilation, and execution. A key contribution of this work is the creation of a complete PolyBench suite rewritten in the MLIR Affine dialect, enabling native experimentation with MLIR-based transformations. Using `Kiriko`, we conduct a comprehensive performance study of Clang, Polly, Pluto, and MLIR Affine across the PolyBench benchmarks, collecting both execution times and hardware performance counters. The results demonstrate the variability of existing polyhedral approaches and highlight the importance of standardized infrastructures for advancing research in loop optimization. Overall, `Kiriko` establishes a reproducible foundation for developing, comparing, and analyzing modern polyhedral optimization techniques.**

***Index Terms*—Polyhedral Optimization, Compiler Optimization, MLIR, LLVM Polly, Pluto, Loop Transformations, Benchmarking, Performance Analysis, PolyBench, Clang**

I. Introduction

The polyhedral model provides a mathematical framework for representing and analyzing loop nests by viewing their iteration spaces as geometric objects in a multidimensional space.[1] Ultimately, this abstraction enables precise reasoning about dependencies among loop iterations, allowing compilers to model transformations as affine mappings over iteration and data spaces.

In fact, polyhedral techniques are particularly effective for optimizing loop-intensive programs, which dominate the runtime of many scientific and high-performance applications. Thus, exploiting the regular structure of loops and array accesses, compilers can safely apply advanced transformations, such as loop interchange, loop tiling, and loop fusion, to improve cache locality, enhance parallelism, and expose opportunities for vectorization and parallel execution [1, 25, 8, 21, 24, 13, 3, 14].

Over the past two decades, several tools have implemented the polyhedral model to perform loop optimizations. Among them, the Pluto optimizer [3] introduced practical scheduling algorithms for automatic parallelization and tiling. The LLVM Polly project [7] brought these techniques to a modern compiler infrastructure, integrating polyhedral analysis directly into LLVM. More recently, the MLIR Affine dialect (part of the MLIR project [11]) has provided a flexible, extensible representation for expressing affine transformations at multiple abstraction levels.

Despite the benefits of polyhedral optimizations, there is no unified infrastructure to systematically explore, compare, and evaluate state-of-the-art polyhedral optimizers. Therefore, each tool is typically evaluated in isolation, using its own compilation pipelines and testing environments. Consequently, this fragmentation makes comparative analysis difficult, as it is challenging to maintain consistent benchmarking conditions and experimental setups.

To address these challenges, we present `Kiriko`, a unified infrastructure for developing and benchmarking polyhedral optimizations. `Kiriko` provides a consistent and automated pipeline to evaluate the performance of LLVM Polly, Pluto, and the MLIR Affine dialect under the same benchmarking conditions using the PolyBench suite. Besides, a key contribution of `Kiriko` is a fully functional PolyBench implementation written in the MLIR Affine dialect, generated with the help of the Polygeist tool [12]. This unified setup enables reproducible side-by-side experiments across different polyhedral frameworks, reducing the complexity of comparative analysis.

The main contributions of this paper are as follows:

- We introduce `Kiriko`, a unified infrastructure for developing and evaluating polyhedral optimizations across different compiler frameworks.
- We release a complete PolyBench Affine implementation to facilitate future research on MLIR-based polyhedral optimization and reproducibility.
- We provide a systematic performance comparison of LLVM Polly, Pluto, and the MLIR Affine dialect using the PolyBench benchmark suite.

[1]Concepts of iteration and data spaces appeared independently in several works from the 1980s and 1990s [23, 22, 5, 4, 2, 9, 16, 6, 10, 17, 18, 20, 15, 19], later forming the *Polyhedral Model*.

II. KIRIKO

A. Design Goals and Motivation

Polyhedral optimizations are being explored on multiple fronts and in various contexts within the compiler community. Thereby, established tools in the polyhedral domain, such as Pluto, Polly, and MLIR Affine, operate within distinct compiler infrastructures and require different input formats and compilation pipelines. As a result, performing fair comparisons or analyses across these tools is time consuming and makes it difficult to maintain equivalent experimental conditions.

Therefore, `Kiriko` was designed as a unieifd, reproducible, and extensible research infrastructure for evaluating and analyzing polyhedral optimizers. Mainly, `Kiriko` design goals are:

- **Unification**, through a common interface that manages benchmarking pipelines for multiple compiler frameworks.
- **Reproducibility**, by providing an easy to use environment to configure compiler flags, benchmark conditions, and optimization pipelines.
- **Automation**, offering an end-to-end workflow to compile, execute, and collect performance metrics automatically.
- **Fairness**, enabling consistent experimental setups that minimize external variability and ensure comparable conditions across compilers.

B. MLIR Affine PolyBench

A key contribution of `Kiriko` is a fully functional version of the PolyBench 3.2 benchmark suite rewritten in the MLIR Affine dialect. As in the original PolyBench/C distribution, this version includes 30 benchmark kernels grouped into the standard categories: linear algebra kernels, linear algebra solvers, datamining, medley, and stencils. With that, `Kiriko` enables researchers to evaluate MLIR-based polyhedral transformations natively and to experiment with novel MLIR passes and dialects under controlled and reproducible conditions.

The MLIR Affine version of PolyBench was generated from the original PolyBench C implementation using Polygeist [12]. However, producing a correct and consistent Affine representation required several manual interventions and preprocessing steps, since Polygeist operates better on fully expanded C code. Thus, we first generated a completely preprocessed version of all PolyBench/C kernels, resolving macro expansions, `#include` directives, and other preprocessor constructs. From this uniform representation, the kernel functions were isolated and translated into MLIR. Following this, the Affine kernels were then manually refined to correct structural issues, normalize index expressions, and ensure compatibility with MLIR optimization and lowering pipelines. The example in figure 1 shows a PolyBench kernel in MLIR Affine.

As a result, `Kiriko` distributes all 30 PolyBench kernels in their Affine form, along with the corresponding preprocessed benchmark driver (*main function*) that handles input generation, timing, and verification. Therefore `Kiriko` ensures that both the MLIR and C pipelines share the same main driver and exhibit fully comparable behavior during benchmarking.

```
func.func @kernel_trmm(%arg0, %arg1 , %arg2, %arg3)
{
  %0 = arith.index_cast %arg0 : i32 to index
  affine.for %arg4 = 1 to %0 {
    affine.for %arg5 = 0 to %0 {
      affine.for %arg6 = 0 to #map(%arg4) {
        %1 = affine.load %arg2[%arg4, %arg6]
        %2 = arith.mulf %arg1, %1
        %3 = affine.load %arg3[%arg5, %arg6]
        %4 = arith.mulf %2, %3
        %5 = affine.load %arg3[%arg4, %arg5]
        %6 = arith.addf %5, %4
        affine.store %6, %arg3[%arg4, %arg5]
      }
    }
  }
  return
}
```

Fig. 1: `TRMM` benchmark Kernel in MLIR Affine (shapes and types omitted).

In addition, `Kiriko` provides full support for compiling and optimizing these benchmarks within MLIR. In this way, users can apply arbitrary MLIR pass pipelines, lower the Affine kernels, and finally generate executable binaries using the LLVM toolchain. This results in end-to-end PolyBench executables implemented and optimized entirely within the MLIR ecosystem, enabling comprehensive evaluation of MLIR Affine and related optimization frameworks.

C. Workflow

Figure 2 illustrates the end-to-end benchmarking workflow implemented in `Kiriko`. The infrastructure integrates MLIR Affine, LLVM Polly, Pluto, and Clang into a unified execution pipeline, ensuring that each optimizer is evaluated under equivalent and reproducible conditions. Despite the inherent differences between the input formats and transformation mechanisms of these compiler frameworks, `Kiriko` abstracts their compilation flows into a high-level and similar sequence of stages.

At this point, we can divide the workflow process into 4 stages: **kernel optimization**, **lowering and object generation**, **benchmark construction**, and **evaluation**. Thus, each polyhedral optimizer was integrated in this structure preserving consistency across experiments.

a) Kernel Optimization Phase: The pipeline begins with two possible sources of kernel code: a C implementation (used for Polly, Pluto, and Clang) or the MLIR Affine implementation (used for the Affine optimizer). `Kiriko` automatically dispatches the kernel to the appropriate optimization backend:

- Kernels written in MLIR Affine are passed through a configurable Affine optimization pipeline.

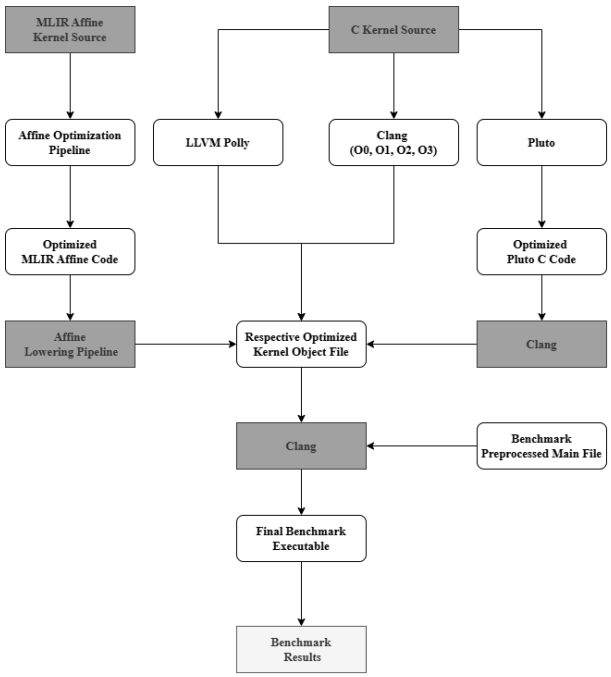

Fig. 2: Kiriko benchmarking workflow integrating MLIR Affine, LLVM Polly, Pluto, and Clang pipelines into a unified framework.

- C kernels destined for analysis by Polly are compiled with Clang into LLVM IR, which is then optimized by Polly.
- C kernels destined for Pluto are optimized and produces an optimized transformed C file.
- Baseline Clang configurations (`O0`, `O1`, `O2`, `O3`) are also compiled directly from the original C kernels.

The output of this phase is an optimized kernel in a representation for subsequent lowering: MLIR Affine IR, LLVM IR, or C source code.

b) Lowering and Object Generation: After optimization, each backend follows its dedicated lowering path until reaching an object file. MLIR Affine kernels uses standard lowering pipelines and subsequently generate object code through the standard LLVM toolchain. Polly optimized kernels already in LLVM IR are compiled directly to object files. Pluto optimized C kernels are compiled with Clang to produce their corresponding object files.

Despite differing intermediate representations, `Kiriko` ensures that all optimizers ultimately produce comparable standalone object files containing only the optimized kernel logic.

c) Final Benchmark: To provide consistent benchmarking conditions, `Kiriko` compiles a preprocessed version of the PolyBench main driver, which handles input generation, timing, and output verification. Then, the framework links this driver with the object file of the optimized kernel to produce a complete executable ready for evaluation.

This step ensures that all kernel variants use equivalent benchmark driver, removing variability from unrelated code.

d) Execution and Evaluation: Once the benchmark executables are generated, `Kiriko` conducts the evaluation phase under controlled and repeatable conditions. After that, users may configure the number of runs for the benchmarks, and `Kiriko` executes all variants of the *same* benchmark sequentially, one backend after another, before moving to the next program in the suite. As a result, this execution strategy minimizes temporal fluctuations in system load and reduces noise when comparing different optimization frameworks.

Moreover, execution times are collected using PolyBench's built-in timing infrastructure, ensuring consistency with prior literature. Additionally, `Kiriko` also integrates `Linux perf` to gather dynamic hardware performance counters, including metrics such as `cpu-cycles`, `branches`, and `cache-misses`. Doing that, these complementary metrics provide deeper insight into the effects of each optimization strategy, enabling analyses that go beyond time performance.

Overall, `Kiriko` automates the complete evaluation loop, from optimization to execution and data collection, and consolidates heterogeneous compiler toolchains into a coherent benchmarking environment. Also, by standardizing runtime conditions and enhancing measurements with low-level performance counters, `Kiriko` enables systematic, fair, and reproducible comparisons among modern polyhedral optimizers.

III. Experimental Setup

As a proof of concept, we evaluated the performance of all optimization tools supported by `Kiriko`: Clang (`-O0` and `-O3`), LLVM Polly, Pluto, and the MLIR Affine Dialect. The Clang `-O0` version was used as the baseline. Along with that, our experiments were conducted on the PolyBench 3.2 benchmark suite, using both its original C version and the MLIR version introduced by `Kiriko`. However, from the 30 PolyBench kernels, the *Symm* benchmark was excluded due to corruption problems involving our chosen MLIR optimization pipeline.

Each benchmark-tool pair was executed 30 times (`SAMPLE_SIZE = 30`). Following the `Kiriko` methodology, the same benchmark is executed for all tools in sequence before moving to the next benchmark, which reduces temporal noise and improves result stability. Furthermore, although several dynamic performance metrics were collected via `perf` (e.g., *cpu-cycles*, *branches*, *cache-misses*), this study focuses primarily on execution time.

A. Test Environment

All experiments were performed under controlled conditions, and `Kiriko` compiled every kernel using the same harness and produced executables under identical runtime conditions. Additionally, we have used the default versions of each tool and avoided specific tuning for each benchmark, keeping the same conditions all the way.

Hardware and System:

- **CPU Model:** Intel(R) Xeon(R) CPU E5-2680 v2 @2.80GHz
- **Cores per Socket(2):** 20

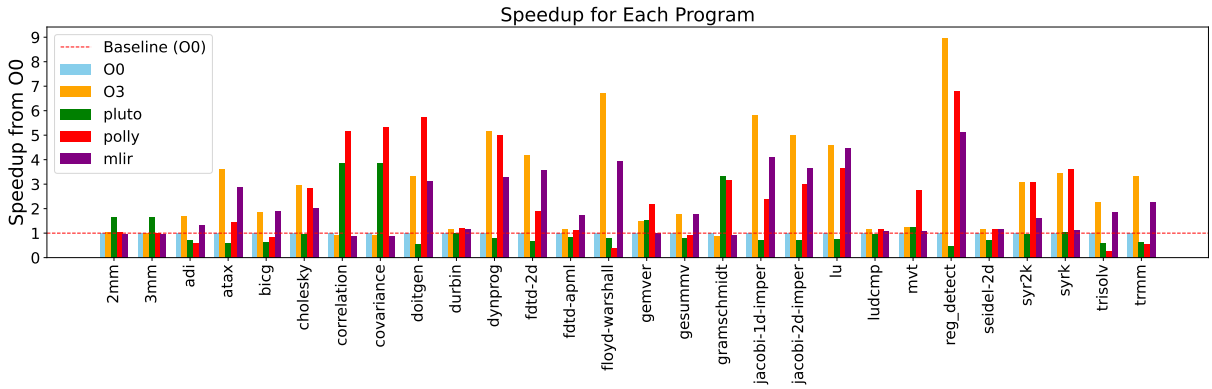

Fig. 3: Side-by-side speedups of arithmetic mean of execution time by benchmark

- **L3 Cache:** 50MiB
- **Memory:** 32 GB (8 x 4GB) DDR3 @ 1333 MT/s

Software Versions:

- **OS Version:** Ubuntu 20.04.6 LTS
- **Clang/LLVM:** version 20.1
- **Polly:** version 20.1
- **Pluto:** version 0.13.0
- **MLIR:** version 20.1
- **Kiriko:** version 1.0.0

B. Tools Configuration

*1) **Clang and Polly**:* All Clang and Polly experiments were executed using their standard optimization flags:

- `clang -O0 kernel.c` (baseline)
- `clang -O3 kernel.c` (standard Clang optimization)
- `clang -O3 -mllvm -polly kernel.c` (LLVM Polly)

*2) **Pluto**:* Pluto requires an optimization step and a compilation step:

- `polycc kernel.c -o pluto-kernel.c` (Pluto optimization)
- `clang -O3 pluto-kernel.c` (final compilation)

*3) **MLIR Affine**:* The MLIR workflow consists of two phases: an optimization pipeline in the Affine dialect, followed by lowering to LLVM IR.

- MLIR Optimization Pipeline

```
mlir-opt kernel.mlir
    -canonicalize
    -cse
    -mem2reg
    -affine-loop-tile=tile-size=32
    -affine-loop-fusion
    -affine-loop-unroll=unroll-factor=4
    -affine-loop-coalescing
    -affine-loop-invariant-code-motion
    -affine-scalrep
    -affine-super-vectorize
```

- MLIR Lowering Pipeline

```
mlir-opt optimized-kernel.mlir
    --lower-affine
    --convert-scf-to-cf
    --convert-cf-to-llvm
    --convert-math-to-funcs
    --convert-math-to-llvm
    --convert-arith-to-llvm
    --convert-func-to-llvm
    --finalize-memref-to-llvm
    -reconcile-unrealized-casts
```

Across all tools, we executed each benchmark under identical build and runtime conditions, using the same harness and measurement infrastructure. Due to this, the uniform setup ensures fair comparisons and minimizes variability not attributable to the optimization techniques themselves.

IV. Results

The 30 executions of each benchmark under every optimization tool in Kiriko's pipeline were aggregated by computing the arithmetic mean runtime for each (program, tool) pair. Thus, we can mitigate the influence of outliers and system fluctuations, ensuring that the reported values reflect the typical performance of each optimization tool accurately. Based on these mean values, we then calculated the normalized speedup relative to the `Clang -O0` baseline, which works as a common point for comparison across the PolyBench benchmarks. Figure 3 presents the speedups achieved by each optimization tool across the PolyBench suite.[2]

Subsequently, in order to summarize the performance across all PolyBench kernels in a single representative metric, we computed the aggregate geometric mean of the programs speedups relative to `Clang -O0`. With that, we can provide a scale-independent notion of the speedups and avoid bias toward benchmarks with large absolute runtimes. As a result,

[2]The `gemm` benchmark was omitted from figure 3 chart for readability, as its values were extreme: O3: 1.045×, Pluto: 1.72×, Polly: 26.98×, MLIR: 0.97×.

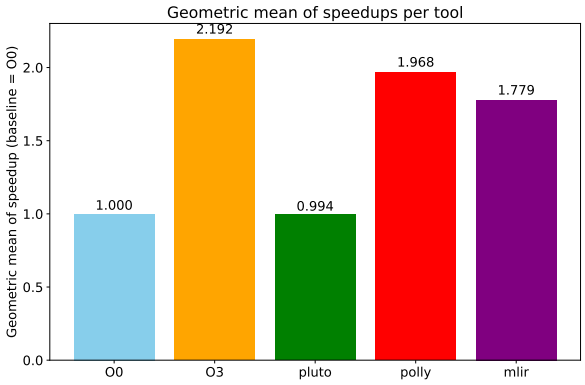

Fig. 4: Side-by-side speedups of geometric mean of execution time by optimization tool

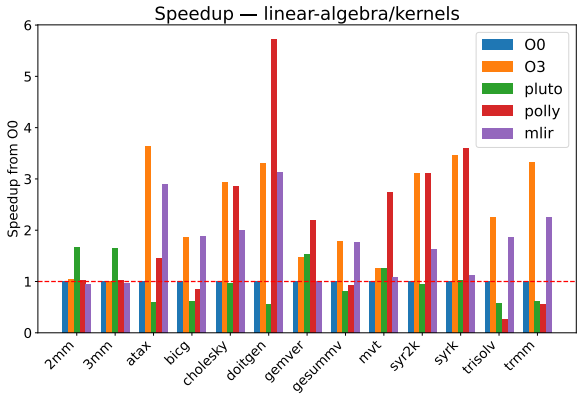

Fig. 5: Speedups for Linear Algebra Kernels

Figure 4 reports the aggregated speedups obtained for `Clang -O3`, Pluto, Polly, and MLIR Affine, revealing how each technology performs when considering the entire benchmark suite.

The speedups presented in Figure 3 reveal substantial variability in how each optimization tool performs across the

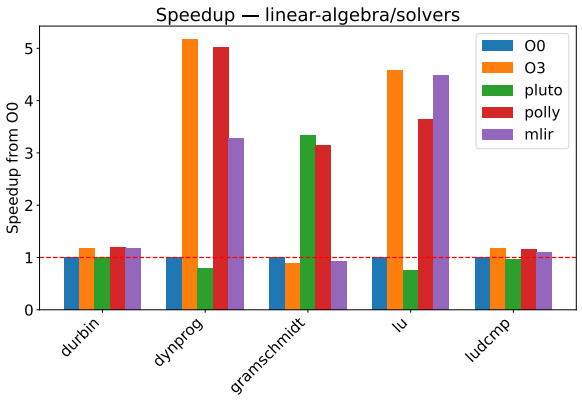

Fig. 6: Speedups for Linear Algebra Solvers

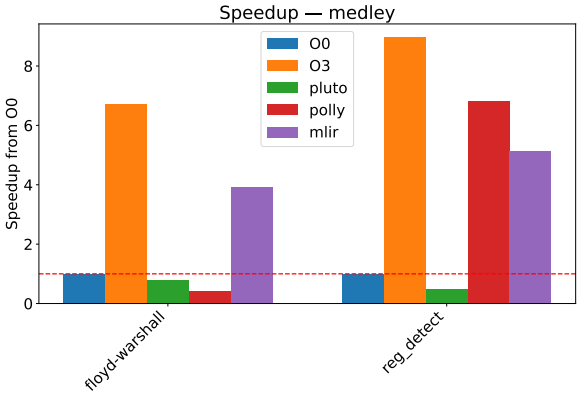

Fig. 7: Speedups for Medley

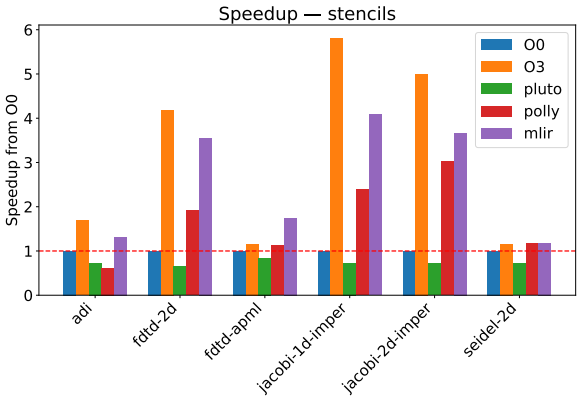

Fig. 8: Speedups for Stencils

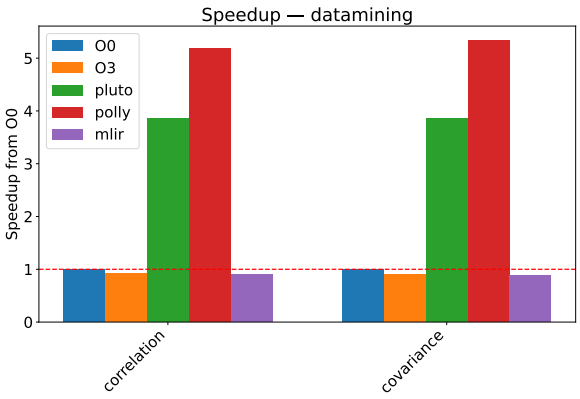

Fig. 9: Speedups for Datamining

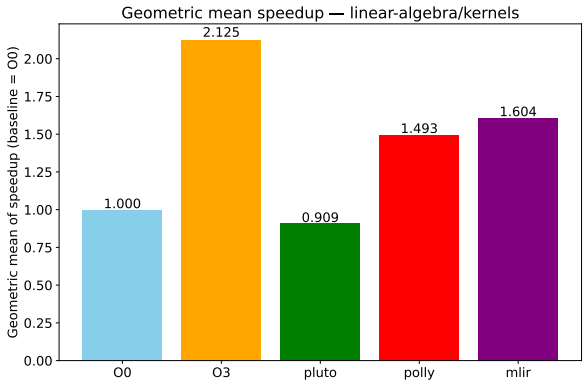

Fig. 10: Geometric Mean of Speedups for Linear Algebra Kernels

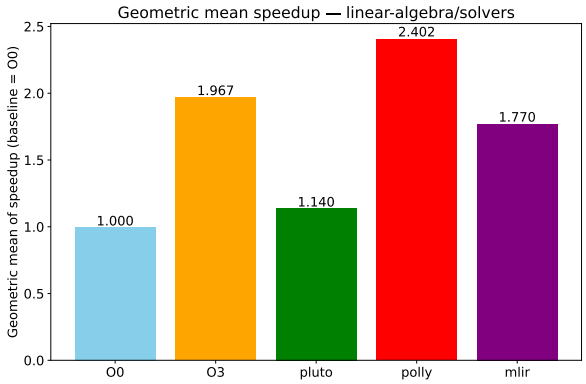

Fig. 11: Geometric Mean of Speedups for Linear Algebra Solvers

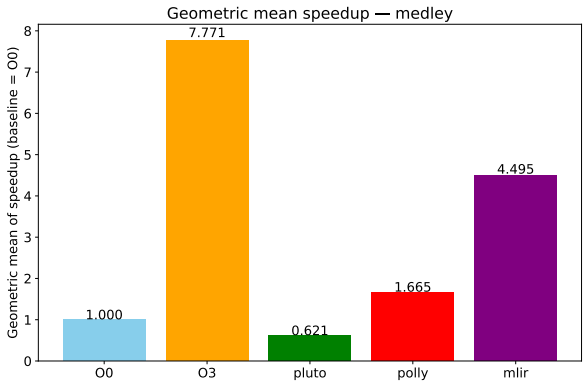

Fig. 12: Geometric Mean of Speedups for Medley

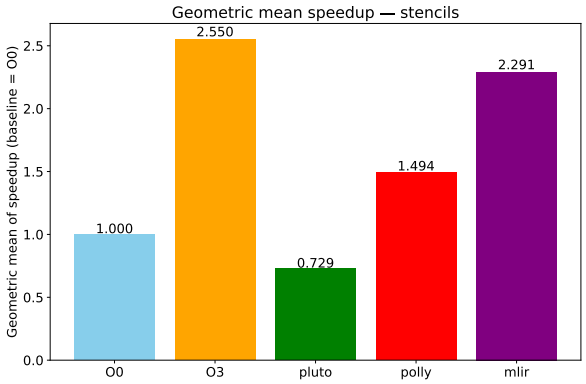

Fig. 13: Geometric Mean of Speedups for Stencils

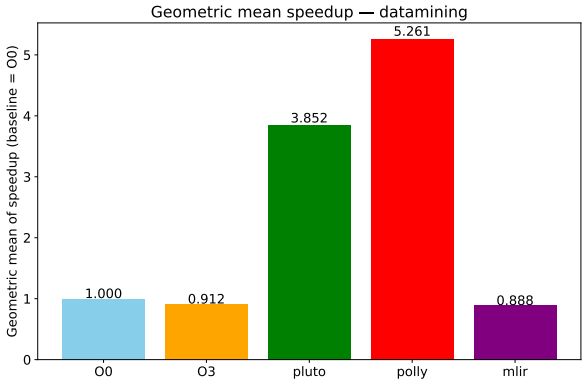

Fig. 14: Geometric Mean of Speedups for Datamining

PolyBench programs. Overall, no tool achieved consistently high speedups for all benchmarks. For example, Polly obtained strong results in benchmarks such as `correlation` and `covariance`, but showed lower speedups in `trmm` and `bicg`. Similarly, both MLIR Affine and `Clang -O3` exhibited cases of notable improvement alongside benchmarks where the performance gains remained limited. In contrast, although Pluto delivered strong speedups in specific cases, such as `2mm` and `covariance`, its results generally remained close to the baseline or fell below it, as observed in `seidel-2d` and `trmm`.

The geometric mean results in Figure 4 reinforce these observations, highlighting the overall performance improvements achieved by the tools. Among them, `Clang -O3` attained the highest geometric mean speedup, followed by Polly and MLIR Affine. Additionally, Pluto's geometric mean remains close to the baseline, consistent with the behavior seen in the first visualization.

To provide a more detailed view of the performance across different computational domains, we grouped the PolyBench kernels into their original categories and computed the speedups and geometric means for each group separately. Hence, the speedup results and aggregated geometric means for each category are shown in the respective figure of the

category.

For the speedup of *linear-algebra/kernels* category, Figure 5, the results display a broad range of speedups among all tools. `Clang -O3` consistently achieves improvements over the baseline, while Pluto, Polly, and MLIR exhibit more varied patterns depending on the specific kernel. As for the geometric mean speedups, Figure 10, this group indicate that `Clang -O3` reaches the highest overall improvement (2.125×), followed by MLIR (1.604×) and Polly (1.493×). Pluto remains close to the baseline with a geometric mean of 0.909×.

In the *linear-algebra/solvers* category, Figure 6, the performance variation across individual kernels is pronounced. Some kernels show substantial speedups for multiple tools, while others remain near the baseline. About the geometric means, Figure 11, Polly achieves the highest aggregated speedup (2.402×), followed by `Clang -O3` (1.967×) and MLIR (1.770×). Pluto again remains near the baseline (1.140×).

Additionally, *medley* category, Figure 7, exhibits the same observed behavior: some tools show high gains, while others remain close to the baseline. The geometric mean results, Figure 12, emphasize this variation: `Clang -O3` achieves the highest aggregated speedup (7.771×), followed by MLIR (4.495×) and Polly (1.665×). Pluto records a geometric mean of 0.671×.

Next, for the *stencils* category, Figure 8, the tools achieve a mix of moderate and high speedups across the kernels. For the geometric mean results, Figure 13, show that `Clang -O3` (2.550×) and MLIR (2.291×) obtain the highest overall improvements, while Polly achieves 1.494× and Pluto records 0.729×.

Finally, in the *datamining* category, Figure 9, the results present similar behavior between the two programs. While some tools achieve significant speedups, others remain near or slightly below the baseline. The geometric mean values, Figure 14, show that Polly obtains the highest aggregated speedup (5.261×), followed by Pluto (3.852×), while `Clang -O3` and MLIR achieve geometric means of 0.912× and 0.888×, respectively.

Overall, the division of the benchmarks into categories reveals substantial variation in performance across computational domains. The relative behavior of each optimization tool depends strongly on the characteristics of the benchmarks within each category, as reflected in both the individual speedups and the aggregated geometric means.

V. Conclusion

This paper introduced `Kiriko`, a unified and reproducible framework designed to support the development, evaluation, and comparison of polyhedral optimizers across heterogeneous compiler frameworks. Thus, by consolidating the compilation flows of Clang, LLVM Polly, Pluto, and MLIR Affine into a single benchmarking environment, `Kiriko` removes the fragmentation traditionally associated with evaluating polyhedral tools. With that purpose, the framework standardizes input preparation, compilation pipelines, and execution conditions, thereby enabling fair and consistent performance comparisons.

A key contribution of this work is the complete and publicly available PolyBench implementation in the MLIR Affine dialect. However, this version not only expands the experimental capabilities of the MLIR ecosystem, but also establishes a foundation for future research in MLIR Affine optimization passes and pipelines. Therefore, `Kiriko` provides a MLIR Affine entry point for experimentation, facilitates reproducibility and supports the development of new optimization passes and dialect extensions.

As a proof of concept, we conducted a side-by-side performance evaluation of four optimization technologies under equivalent conditions using the `Kiriko` framework. The results highlight that the effectiveness of polyhedral optimizations varies significantly across benchmarks and computational domains, reinforcing the need for systematic and controlled experimentation frameworks such as `Kiriko`. Moreover, the geometric mean analyses demonstrate how each tool behaves on average across the full suite and within specific categories, further illustrating the role of `Kiriko` as a platform for nuanced and domain-aware evaluation.

Overall, `Kiriko` provides a robust, extensible, and automated environment for advancing research in polyhedral optimization. Furthermore, `Kiriko` offers a unified workflow, comparable pipelines, and MLIR-native benchmark kernels, lowering the barrier to experimentation and paves the way for more comprehensive studies on performance, correctness, and transformation design in modern compiler infrastructures. Additionally, future work includes incorporating additional MLIR dialects, expanding support for heterogeneous hardware targets, and extending the framework metric collector and enabling parallel execution models.

References

[1] Khaled Abdelaal and Martin Kong. “Tile size selection of affine programs for GPGPUs using polyhedral cross-compilation”. In: *Supercomputing*. New York, USA: ACM, 2021, pp. 13–26.

[2] S. Amarasinghe and M. Lam. “Communication Optimization and Code Generation for Distributed Memory Machines”. In: *PLDI*. Albuquerque, N.M.: ACM Press, June 1993, pp. 126–138.

[3] Uday Bondhugula et al. “A practical automatic polyhedral parallelizer and locality optimizer”. In: *PLDI*. PLDI ’08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 101–113. ISBN: 9781595938602. DOI: 10.1145/1375581.1375595. URL: https://doi.org/10.1145/1375581.1375595.

[4] Paul Feautrier. “Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time”. In: *International Journal of Parallel Programming* 21.5 (1992), pp. 313–347.

[5] Paul Feautrier. “Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time”. In: *International Journal of Parallel Programming* 21.6 (1992), pp. 389–420.

[6] J. A. B. Fortes and D. Moldovan. “Data Broadcasting in Linearly Scheduled Array Processors”. In: *Symposium on Computer Architecture*. 1984, pp. 224–231.

[7] TOBIAS GROSSER, ARMIN GROESSLINGER, and CHRISTIAN LENGAUER. “POLLY — PERFORMING POLYHEDRAL OPTIMIZATIONS ON A LOW-LEVEL INTERMEDIATE REPRESENTATION”. In: *Parallel Processing Letters* 22.04 (2012), p. 1250010. DOI: 10.1142/S0129626412500107. eprint: https://doi.org/10.1142/S0129626412500107.

[8] Rajiv Gupta et al. “Compilation techniques for parallel systems”. In: *Parallel Computing* 25.13-14 (1999), pp. 1741–1783.

[9] F. Irigoin and R. Triolet. “Supernode Partitioning”. In: *POPL*. ACM. Jan. 1988, pp. 319–328.

[10] M. S. Lam. *A Systolic Array Optimizing Computer*. ISBN:-13: 978-14612-8961-6. Kluwer Academic (Springer), 1989. DOI: 10.1007/978-1-4613-1705-0.

[11] Chris Lattner et al. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[12] William S. Moses et al. “Polygeist: Raising C to Polyhedral MLIR”. In: *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’21. Atlanta, GA, USA: IEEE Press, 2024, pp. 45–59. ISBN: 9781665442787. DOI: 10.1109/PACT52795.2021.00011. URL: https://doi.org/10.1109/PACT52795.2021.00011.

[13] Tiago Daniel Costa Oliveira. “Analysis and Comparison of Automatic Parallelization Tools”. MA thesis. Instituto Politecnico do Cavado e do Ave (Portugal), 2023.

[14] Louis-Noël Pouchet et al. “Hybrid iterative and model-driven optimization in the polyhedral model”. PhD thesis. INRIA, 2009.

[15] W. Pugh. “A practical algorithm for exact array dependence analysis”. In: *Commun. ACM* 35.8 (1992), pp. 102–114. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/135226.135233.

[16] P. Quinton and V. Van Dongen. “The Mapping of Linear Recurrence Equations on Regular Arrays”. In: *Journal of VLSI Signal Processing* 1.2 (1989), pp. 95–113.

[17] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. “On Synthesizing Systolic Arrays from Recurrence Equations with Linear Dependencies”. In: *Foundations of Software Technology and Theoretical Computer Science*. New Delhi, India: Springer Verlag, LNCS 241, Dec. 1986, pp. 488–503.

[18] S. V. Rajopadhye. “Synthesis, Optimization and Verification of Systolic Architectures”. PhD thesis. Salt Lake City, Utah 84112: University of Utah, Dec. 1986.

[19] J. Ramanujam and P. Sadayappan. “Nested Loop Tiling for Distributed Memory Multiprocessors”. In: *Distributed Memory Computer Conference, V*. Charleston, Apr. 1990, pp. 1088–1096.

[20] R. Schreiber and J. Dongarra. *Automatic blocking of nested loops*. Tech. rep. 90.38. NASA Ames Research Center: RIACS, Aug. 1990.

[21] Nicolas Tollenaere. “Decoupling the optimization space of tensor computation for a better understanding of performance on Intel CPU”. PhD thesis. Université Grenoble Alpes [2020-....], 2022.

[22] M. Wolf and M. Lam. “Loop transformation theory and an algorithm to maximize parallelism”. In: *Transactions on Parallel and Distributed Systems* 2.4 (Oct. 1991), pp. 452–471.

[23] M. J. Wolfe. “Iteration space tiling for memory hierarchies”. In: *Parallel Processing for Scientific Computing (SIAM)* (1987), pp. 357–361.

[24] Yuanrui Zhang. *Cache-aware application parallelization and optimization for multicores*. The Pennsylvania State University, 2011.

[25] Oleksandr Zinenko et al. “Unified polyhedral modeling of temporal and spatial locality”. PhD thesis. Inria Paris, 2017.