An MLIR Version of the PolyBench Benchmark Suite for High-Performance Computing

Lucas Victor Costa[1], **José Wesley Magalhães**[1],
Michael Canesche[1], **Fernando Pereira**[2]

[1]Cadence Design Systems
30110-042 – Belo Horizonte – MG – Brazil

[2]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brazil

{lcosta, jwesley, canesche}@cadence.com, 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 create a complete PolyBench suite rewritten in the MLIR Affine dialect, PolyBench-MLIR, enabling native experimentation with MLIR-based transformations. We further 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. Using* `Kiriko` *and PolyBench-MLIR, we conduct a comprehensive performance study of Clang, Polly, Pluto, and MLIR Affine across the PolyBench benchmarks. The results demonstrate the variability of existing polyhedral approaches across computational domains and highlight the importance of standardized infrastructures for advancing research in loop optimization. Overall,* `Kiriko` *and the PolyBench-MLIR suite establish a reproducible foundation for developing, comparing, and analyzing modern polyhedral optimization techniques.*

1. Introduction

PolyBench [Pouchet et al. 2012] has long served as a standard benchmark suite in the high-performance computing (HPC) community to evaluate loop-intensive programs and polyhedral optimization techniques. Its collection of kernels across domains such as linear algebra, stencils, and data mining has allowed researchers to systematically assess performance gains across different compilation toolchains [Herten et al. 2026, Tørring et al. 2023, Queiroz et al. 2023, Wu et al. 2022, Weinhardt 2022, Abella-González et al. 2021, Sreenivasan et al. 2019, Bondhugula et al. 2008].

Recently, Multi-Level Intermediate Representation (MLIR) [Lattner et al. 2021] has emerged as a state-of-the-art compiler infrastructure, providing modular, dialect-based abstractions that have quickly become the standard for building domain-specific compilers and high-performance machine learning frameworks.

Unfortunately, currently there is no native version of PolyBench for MLIR. Creating such a benchmark suite is far from trivial: automatic C-to-MLIR translation tools like Polygeist [Moses et al. 2024] cannot fully bridge the gap without manual intervention, requiring lower-level IR structural refinements, index expression normalizations, and preprocessor expansions to produce a valid MLIR Affine representation.

To solve this problem, this paper introduces PolyBench-MLIR, a complete, fully functional PolyBench 3.2 suite [Yuki 2014] rewritten in the MLIR Affine dialect, alongside `Kiriko`, an automated benchmarking infrastructure designed for fair, reproducible cross-toolchain evaluation. This native MLIR version enables the high-performance computing community to systematically experiment with advanced compiler optimizations across the MLIR ecosystem. These optimizations range from high-level loop transformations like tiling and fusion in the `linalg` dialect to automatic kernel launch generation, thread-block mapping, and memory hierarchy management in the `gpu` dialect.

We used this suite to conduct a comprehensive empirical evaluation of several compiler optimization toolchains focused on polyhedral optimizations in Section 4. Our findings reveal significant performance variability across computational domains: while `Clang -O3` achieves the highest geometric mean speedup overall ($2.19\times$), individual polyhedral optimizers excel in specific domains. For instance, LLVM Polly [Grosser et al. 2012] achieves top performance in linear algebra solvers ($2.40\times$) and datamining ($5.26\times$), whereas MLIR Affine delivers strong competitive speedups in stencils ($2.29\times$) and medley kernels ($4.50\times$).

The main contributions of this paper are as follows:

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

2. The PolyBench-MLIR Collection

The PolyBench-MLIR collection is a fully functional version of the PolyBench 3.2 benchmark suite rewritten in the MLIR Affine dialect [LLVM Project. 2026]. We selected the Affine dialect because it provides a structured representation of loop nests, memory accesses, and affine constraints, enabling dependence analysis and a broad range of polyhedral loop transformations within the MLIR ecosystem. In addition, the dialect serves as the foundation for several optimization passes in MLIR, making it a ideal target to express polyhedral benchmarks. As in the original PolyBench/C distribution, our 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.

We generated PolyBench-MLIR from the original PolyBench C implementation using Polygeist [Moses et al. 2024]. However, producing a correct and consistent Affine

```
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
}
```

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

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, we isolated the kernel functions and translated them into MLIR. Following this, we manually refined the affine kernels 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.

PolyBench-MLIR distributes all 30 PolyBench kernels in their Affine form, along with a corresponding preprocessed benchmark driver (*main function*) that handles input generation, timing, and verification. This suite is publicly available [1].

3. Kiriko

Polyhedral optimizations have been explored on multiple fronts and in various contexts within the compiler community. Thereby, established tools in the polyhedral domain, such as Pluto [Bondhugula et al. 2008], Polly [Grosser et al. 2012], and MLIR Affine [LLVM Project. 2026], operate within distinct infrastructures and test environments, besides requiring different input formats and compilation pipelines. Consequently, this fragmentation makes comparative analysis difficult, as it is challenging to maintain consistent benchmarking conditions and experimental setups.

To overcome these challenges, we designed `Kiriko` as a unified, 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.

[1] `https://github.com/lac-dcc/Kiriko/`

- **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.

3.1. 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.

We 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.

Kernel Optimization. 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.
- 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.

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. In addition, `Kiriko` provides full support for users to apply arbitrary MLIR pass pipelines, lower the Affine kernels, 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.

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

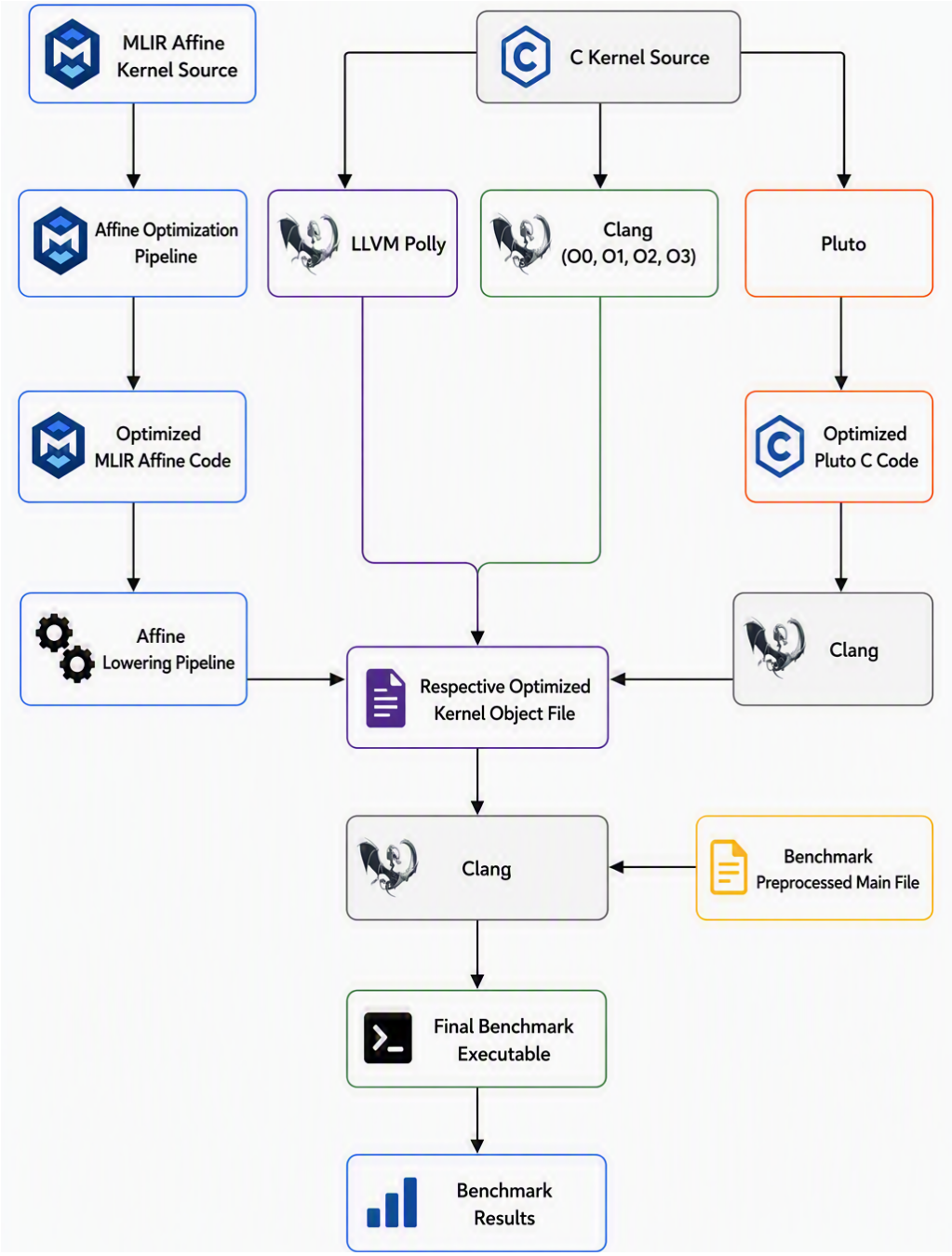

Figura 2. `Kiriko` benchmarking workflow integrating MLIR Affine, LLVM Polly, Pluto, and `Clang` pipelines into a unified framework.

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

Execution and Evaluation. Once we generate each benchmark executable, `Kiriko` conducts the evaluation phase under controlled and repeatable conditions. 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. This execution strategy minimizes temporal fluctuations in system

load and reduces noise when comparing different optimization frameworks.

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

Toolchains. Currently, `Kiriko` supports the following optimizing toolchains:

- **`Clang`** [Lattner and Adve 2004]: A production-grade C/C++ compiler widely used in both academia and industry.
- **Polly** [Grosser et al. 2012]: An LLVM subproject that applies polyhedral analysis and transformations to optimize static-control loop nests during compilation.
- **Pluto** [Bondhugula et al. 2008]: A source-to-source polyhedral optimizer that automatically performs loop transformations such as tiling, fusion, and parallelization to improve locality and parallelism.
- **MLIR Affine** [LLVM Project. 2026]: The MLIR dialect that represents affine loop nests and memory accesses using polyhedral abstractions.

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.

4. Evaluation

In this section, we evaluate the overall speedup enabled by `Kiriko` supported optimization tools. We first provide a full description of the experimentl setup (Section 4.1) followed by an overall speedup analysis (Section 4.2.1) and a breakdown of each tool's performance on different computation domains (Section 4.2.2).

4.1. Experimental Setup

We evaluate the performance of all optimization tools supported by `Kiriko`: `Clang`, LLVM Polly, Pluto, and the MLIR Affine Dialect. We use `Clang -O0` as the baseline, and evaluate `Clang -O3`. For the benchmarks, we evaluate the entire PolyBench 3.2 benchmark suite, using both its original C version and PolyBench-MLIR. Because corruption issues arose in our selected MLIR optimization pipeline, we excluded the *Symm* benchmark from the evaluation. The resulting benchmark suite contains 29 kernels.

Software. We use Clang/LLVM version 20.1, Polly version 20.1, Pluto version 0.13.0, MLIR version 20.1 and `Kiriko` version 1.0.0. The operating system is Ubuntu 20.04.6 LTS.

Hardware. We evaluate on a 20-core Intel(R) Xeon(R) CPU E5-2680 v2 CPU at 2.80 GHz with 32 GB (8x4 GB) (DDR3) memory at 1333 MT/s and 50 MiB of L3 Cache.

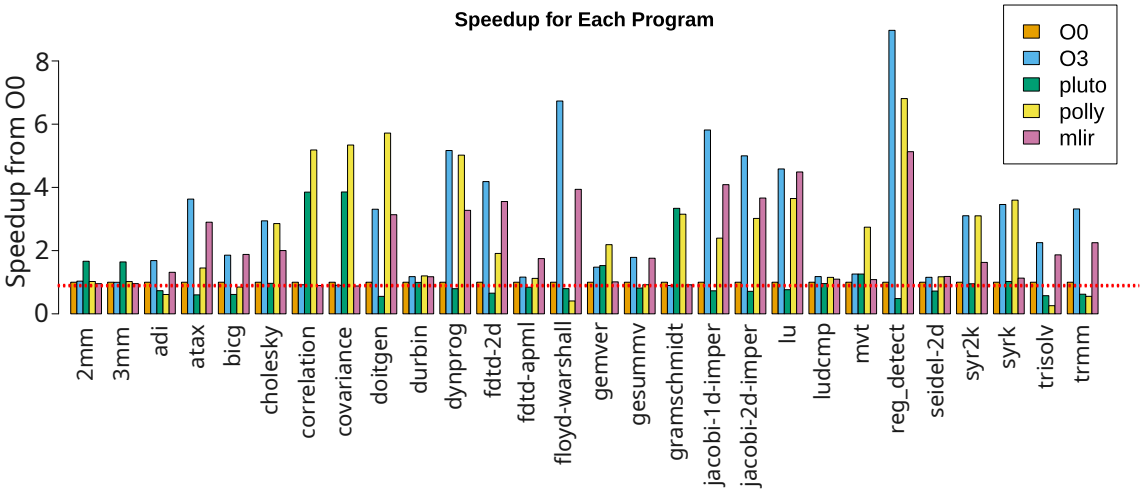

Figura 3. Side-by-side speedups of arithmetic mean of execution time by benchmark

Methodology. We perform all the experiments under controlled conditions, i.e., we compiled every benchmark with `Kiriko` using the same harness and produced binaries under identical runtime conditions. Additionally, we use the default versions of each tool and avoided benchmark-specific tuning. We execute each $\langle benchmark, tool \rangle$ 30 times. Following the `Kiriko` methodology, we execute the same benchmark for all tools in sequence before moving to the next benchmark, which reduces temporal noise and improves result stability.

Although several dynamic performance metrics can be collected (e.g., *cpu-cycles*, *branches*, *cache-misses*), this study primarily focuses on execution time. We measure what are the performance gains (speedup) achieved by each tool on the entire suite. To compute speedup, we average the runtime of the 30 executions for each $\langle benchmark, tool \rangle$ pair across all benchmarks and tools in `Kiriko`'s pipeline. This mitigates outliers and system fluctuations, ensuring that the reported values accurately reflect the performance of each optimization tool. We report speedup as the ratio of these average values relative to the baseline `Clang -O0`.

4.2. Results

4.2.1. Speedup

Figure 3 presents the speedups achieved by each optimization tool across the entire PolyBench suite.[2] These results reveal substantial variability in the performance of each optimization tool. `Clang -O3` delivers the highest speedup in 48% of the benchmarks, followed by Polly (24%), and Pluto (10%). MLIR Affine is the fastest optimizer in the `fdtd-apml` benchmark, and in four benchmarks, the differences among the optimizers were negligible.

Overall, no tool achieved consistently high speedups for all benchmarks. For example, Polly obtained strong results in benchmarks such as `correlation` and

[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×.

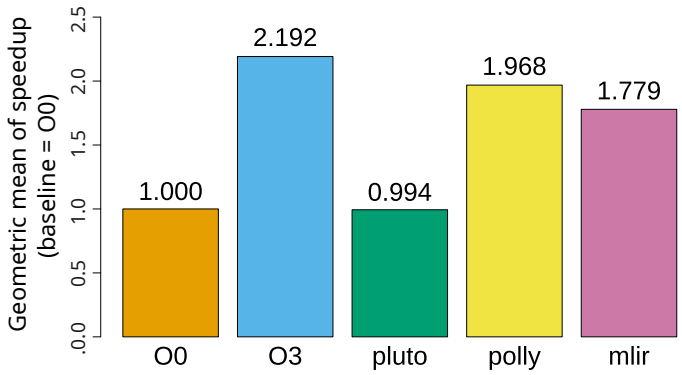

Figura 4. Side-by-side speedups of geometric mean of execution time by optimization tool

`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. Pluto delivered strong speedups in specific cases, such as `2mm` and `covariance`. However, its results generally remained close to the baseline and in two cases, `seidel-2d` and `trmm`, we observe performance degradation compared to the baseline.

We further compute the aggregate mean of speedup values to summarize the performance across all PolyBench kernels in a single representative metric. This provides a scale-independent notion of the speedups and avoid bias toward benchmarks with large absolute runtimes. Figure 4 reports the aggregated speedups obtained for each evaluated tool.

The results in Figure 4 reinforce the observations from Figure 3. Among the evaluated tools, `Clang -O3` achieved the highest geometric mean speedup, followed by Polly and MLIR Affine. This is consistent with the frequency in which these tools emerged as the best-performing optimizer. Pluto's geometric mean remains close to the baseline. It is worth to highlight that although MLIR Affine achieved the highest speedup only in a single benchmark, its geomean speedup is higher than Pluto's. This due to the performance degradation introduced by Pluto in some benchmarks as previously discussed.

4.2.2. Speedup by Category

To provide a more detailed view of the performance across different computational domains, we grouped the PolyBench kernels into their original categories, namely *linear-algebra/kernels*, *linear-algebra/solvers*, *medley*, *stencils*, and *datamining*. We computed the speedups and geometric mean for each group separately. Figures 5 and 6 show these results.

For the *linear-algebra kernels* category, `Clang -O3` overall achieves improvements over the baseline, while Pluto, Polly, and MLIR exhibit varied kernel-specific patterns as shown in Figure 5(a). This is reflected in the geometric mean values, as shown in Figure 6(a), in which `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

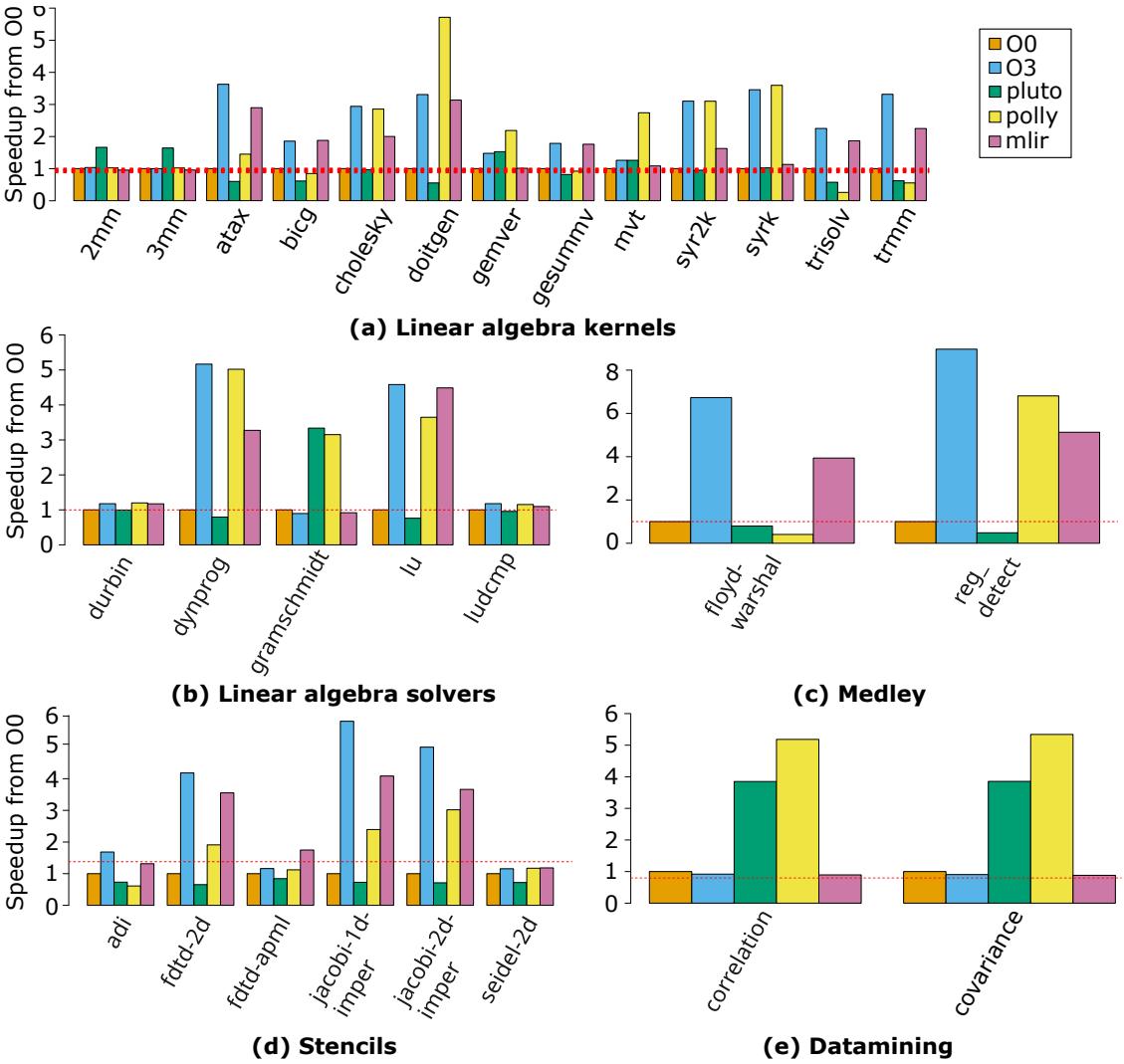

Figura 5. Speedups achieved by each optimization tool across the PolyBench categories.

a geometric mean of 0.909×.

Figure 5(b) shows the performance variation across individual kernels in the *linear-algebra solvers* group. Multiple tools achieve substantial speedups in the `dynprog`, `gramschmidt`, and `lu` kernels, while all tools remain near the baseline in `durbin` and `ludcmp`. Figure 6(b) shows that 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×).

In the *medley* category, Figure 5(c) shows that `Clang -O3` and MLIR Affine achieve significant speedup over the baseline in both benchmarks, while Polly performs well on `reg_detect` but poorly on `floyd-warshall`, and Pluto does not outperform the baseline in any benchmark. In terms of geometric mean, `Clang -O3` achieves the highest aggregated speedup (7.771×), followed by MLIR (4.495×) and Polly (1.665×), while Pluto is slower than the baseline with a geometric mean of 0.671× as illustrated in Figure 6(c).

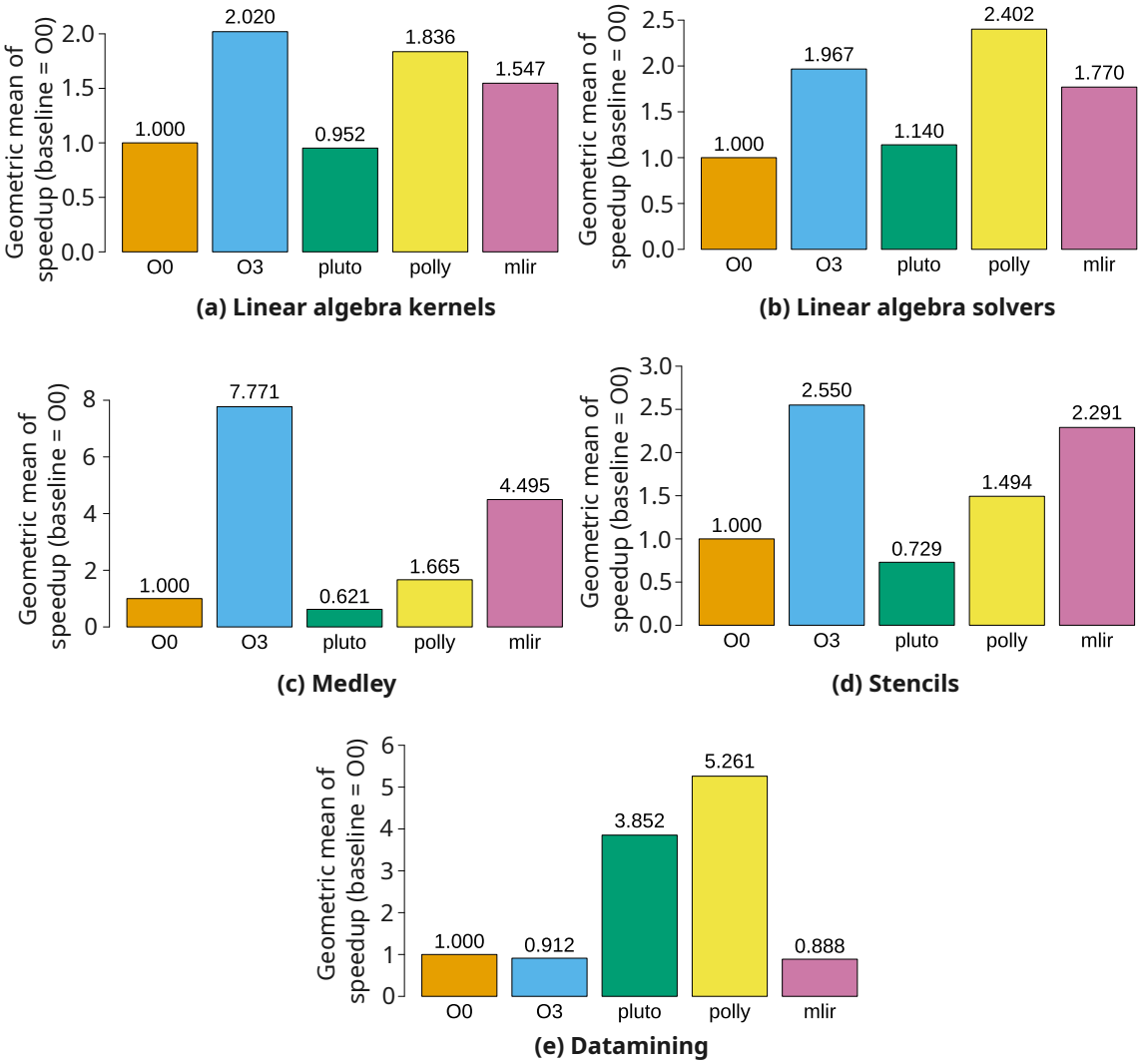

Figura 6. Geomean speedup achieved by each optimization tool across the Poly-Bench categories.

The majority of the optimization tools achieve good speedup values on the *stencil* category on Figure 5(d). `Clang -O3` and MLIR Affine are able to outperform the baseline for all the programs in the group, which the first being the best optimizer in 66% of the cases. Polly delivers performance gains from 2× to 3× in half of the *stencil* kernels, but it is slower than the baseline in the `adi` kernels. As well as in the `medley` category, Pluto does not deliver speedup gains in any benchmark. Figure 6(d), shows that `Clang -O3` and MLIR Affine obtain the highest overall improvements (2.550×) and MLIR (2.291×), respectively, while Polly achieves 1.494× and Pluto 0.729×.

Finally, Figure 5(e) shows the performance gains in the *datamining* category. Unlike the previous groups, `Clang -O3` and MLIR Affine perform poorly and are not able optimize programs, while Pluto delivers 4× speedup in both benchmarks. In this category, Polly is the best overall optimizer, obtaining 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. Figure 6(e) depicts the aggregated speedup values for *datamining* benchmarks.

Overall, we observe substantial variation in performance across computational domains. This result indicates that the performance of each optimization tool depends strongly on the characteristics of the benchmarks within each category, as reflected by both the individual speedups and the aggregated geometric means. For example, `Clang -O3` and MLIR Affine achieve strong results in most of the categories, while Pluto remains close to the baseline in those same groups. Yet, that trend does not hold for the entire suite, as these tools exhibit the opposite behavior in *datamining* programs.

5. Related Work

Benchmark suites have long played a central role in evaluating compiler and architecture innovations for high-performance computing. Well-established collections such as the NAS Parallel Benchmarks [Bailey et al. 1991] and SPEC CPU [Bucek et al. 2018], have been widely adopted to study the impact of hardware and software innovations under controlled conditions. Rodinia [Che et al. 2009] introduced a diverse set of applications targeting multi-core CPUs and GPUs, selected to cover the parallel communication and synchronization patterns of Berkeley's dwarf taxonomy, and has since become a standard reference for heterogeneous-system characterization. Parboil [Stratton et al. 2012] similarly collected throughput-oriented scientific and commercial kernels, deliberately including multiple implementations of the same algorithm at different optimization levels so that compilers and architectures could be compared on equal algorithmic footing. Such works have inspired the creation of other collections of benchmarks, such as Lee et al.'s OpenCL suite [Lee et al. 2010]. While these suites remain widely used, neither targets the polyhedral model specifically, nor do they provide a native MLIR representation of their kernels; adapting them to evaluate MLIR-based polyhedral transformations would require the same manual translation effort that motivated our PolyBench-MLIR port.

Within the polyhedral community, PolyBench itself [Pouchet et al. 2012] has served as the de facto benchmark for evaluating loop transformation techniques. However, unlike other benchmark suites that have evolved alongside emerging programming models and execution environments, PolyBench has historically remained centered on C-based implementations, creating challenges for evaluating modern compiler infrastructures such as MLIR. Consequently, comparisons across Pluto, Polly, and MLIR Affine have historically relied on ad hoc, per-tool scripts rather than a shared infrastructure.

Closer to our MLIR-focused contribution, SODA-OPT [Agostini et al. 2022] proposes an MLIR-based compiler flow for hardware acceleration that outlines and lowers kernels (including PolyBench kernels) into progressively lower MLIR dialects to target accelerator synthesis, demonstrating the value of MLIR Affine as an intermediate representation for optimization studies. However, its focus is accelerator generation rather than cross-toolchain CPU benchmarking, and it does not provide a general-purpose, publicly available MLIR Affine translation of the full PolyBench suite. More broadly, tools such as Polygeist [Moses et al. 2024] enable automatic C-to-MLIR raising and have been used to bootstrap MLIR versions of existing C codebases, but, as discussed in Section 2, raw Polygeist output requires substantial manual refinement before it is usable as a stable, reproducible benchmark artifact.

Kiriko differs from these efforts by combining (i) a complete, manually validated PolyBench Affine translation with (ii) a unified benchmarking harness that places MLIR

Affine, LLVM Polly, Pluto, and Clang under identical compilation and execution conditions. To our knowledge, no previous work provides this combination of benefits.

6. Conclusion

This paper developed PolyBench-MLIR, a complete and publicly available PolyBench implementation in the MLIR Affine dialect. 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. Another contribution is `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. Besides, by providing seamless MLIR integration, `Kiriko` provides an 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 PolyBench-MLIR and `Kiriko`. 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. 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.

Together, PolyBench-MLIR and `Kiriko` provide a robust, extensible, and automated environment for advancing research in polyhedral optimization, offering a unified workflow, comparable pipelines, and MLIR-native benchmark kernels. This lowers 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 targeting additional MLIR dialects, expanding support for heterogeneous hardware targets, and extending the framework metric collector and enabling parallel execution models.

6.1. Software

PolyBench-MLIR and `Kiriko` are publicy available at `https://github.com/lac-dcc/Kiriko/` under Apache 2.0 License.

Referências

Abella-González, M. Á., Carollo-Fernández, P., Pouchet, L.-N., Rastello, F., and Rodríguez, G. (2021). Polybench/python: benchmarking python environments with polyhedral optimizations. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 59–70, New York, NY, USA. Association for Computing Machinery.

Agostini, N. B., Curzel, S., Kaeli, D., and Tumeo, A. (2022). Soda-opt an mlir based flow for co-design and high-level synthesis. In *CF*, page 201–202, New York, NY, USA. Association for Computing Machinery.

Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. (1991). The nas parallel benchmarks. *Int. J. High Perform. Comput. Appl.*, 5(3):63–73.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008). A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, PLDI '08, page 101–113, New York, NY, USA. Association for Computing Machinery.

Bucek, J., Lange, K.-D., and v. Kistowski, J. (2018). Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA. Association for Computing Machinery.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, IISWC '09, page 44–54, USA. IEEE Computer Society.

Grosser, T., Groesslinger, A., and Lengaur, C. (2012). Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010.

Herten, A., Pearce, O., and Guimarães, F. S. (2026). An hpc benchmark survey and taxonomy for characterization. *The International Journal of High Performance Computing Applications*, 40(1):42–51.

Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. (2021). MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14.

Lee, J., Kim, J., Seo, S., Kim, S., Park, J., Kim, H., Dao, T. T., Cho, Y., Seo, S. J., Lee, S. H., Cho, S. M., Song, H. J., Suh, S.-B., and Choi, J.-D. (2010). An opencl framework for heterogeneous multicores with local memory. In *PACT*, PACT '10, page 193–204, New York, NY, USA. Association for Computing Machinery.

LLVM Project. (2026). Mlir affine dialect. `https://mlir.llvm.org/docs/Dialects/Affine/`.

Moses, W. S., Chelini, L., Zhao, R., and Zinenko, O. (2024). Polygeist: Raising c to polyhedral mlir. In *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques*, PACT '21, page 45–59. IEEE Press.

Pouchet, L.-N. et al. (2012). Polybench: The polyhedral benchmark suite.

Queiroz, F., Damasceno, E., and Amaris, M. (2023). Predição de consumo de energia de aplicações openmp multithreads. In *Escola Regional de Alto Desempenho Norte 2*

(ERAD-NO2) e Escola Regional de Aprendizado de Máquina e Inteligência Artificial Norte 2 (ERAMIA-NO2), pages 21–24. SBC.

Sreenivasan, V., Javali, R., Hall, M., Balaprakash, P., Scogland, T. R. W., and de Supinski, B. R. (2019). A framework for enabling openmp autotuning. In *OpenMP: Conquering the Full Hardware Spectrum: 15th International Workshop on OpenMP, IWOMP 2019, Auckland, New Zealand, September 11–13, 2019, Proceedings*, page 50–60, Berlin, Heidelberg. Springer-Verlag.

Stratton, J. A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G. D., and Hwu, W.-m. W. (2012). Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127(7.2):27.

Tørring, J. O., van Werkhoven, B., Petrovč, F., Willemsen, F.-J., Filipovič, J., and Elster, A. C. (2023). Towards a benchmarking suite for kernel tuners. In *2023 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 724–733. IEEE.

Weinhardt, M. (2022). An analysis of mapping polybench kernels to hpc cgras. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 647–654.

Wu, X., Kruse, M., Balaprakash, P., Finkel, H., Hovland, P., Taylor, V., and Hall, M. (2022). Autotuning polybench benchmarks with llvm clang/polly loop optimization pragmas using bayesian optimization. *Concurrency and Computation: Practice and Experience*, 34(20):e6683.

Yuki, T. (2014). Understanding polybench/c 3.2 kernels. In *International workshop on polyhedral compilation techniques (IMPACT)*, pages 1–5.