Bringing Presburger Arithmetic to MLIR with FPL

Arjun Pitchanathan
University of Edinburgh
arjun.pitchanathan@ed.ac.uk

Kunwar Shaanjeet Singh Grover
IIIT Hyderabad
kunwar.shaanjeet@students.iiit.ac.in

Michel Weber
ETH Zurich
webmiche@student.ethz.ch

Tobias Grosser
University of Edinburgh
tobias.grosser@ed.ac.uk

Abstract

Presburger arithmetic provides the mathematical core for the polyhedral compilation techniques that drive analytical cache models, loop optimization for ML and HPC, formal verification, and hardware design. Despite many efforts to use Presburger arithmetic in production compilers, the most powerful polyhedral compilation techniques have thus far been confined to optional extensions of the compilation flow as traditional Presburger libraries were developed as standalone projects that were not part of the core compiler toolchain. After developing FPL, a fast new Presburger library, we worked with the LLVM community to make Presburger arithmetic an integral part of the production-focused MLIR compiler framework. We show how developers can use FPL in MLIR and report on our experience in working with the LLVM/MLIR community on integrating FPL into MLIR. With FPL now a native, performant, and approachable component of the LLVM ecosystem, compiler developers can co-develop the Presburger library and polyhedral transformations with ease. As a result, we expect that scientists and compiler engineers will now be able to jointly develop even more IMPACTful polyhedral optimizations.

1 Introduction

Polyhedral compilation [19] based on Presburger arithmetic is used for performance optimization in high-performance computing and machine learning [1, 3, 6, 18], formal verification [13], cache modelling [8], the derivation of data movement bounds [14], and configurable computing [16]. Over the last thirty years, several libraries provided mathematical foundations for polyhedral compilation. Omega [9] and Polylib [11], developed many of the early ideas and inspired the design of polyhedral math libraries. The Parma Polyhedral Library (PPL) [2] was the first polyhedral library to be used in GCC, as part of Graphite [17]; isl further improved on its robustness, provided the foundations for LLVM/Polly [5] and Tensor Comprehensions [18], among others, and is considered state-of-the-art in polyhedral libraries. Finally, VPL [4] demonstrated formally verified implementations for certain operations on sets of rationals.

As a Presburger library for LLVM/Polly [6], GCC [17], and many other tools, isl outgrew its mathematical roots quickly and increasingly incorporated broader compiler concepts. In particular, isl offered code generation facilities [7] as well as higher-level code representations such as schedule trees [20]. While schedule trees made polyhedral compilers more powerful, they were incompatible with the design and needs of LLVM [10]. MLIR then generalized schedule trees with nested regions, and used these with the Affine dialect to create a simplified loop model that integrates well with the broader LLVM ecosystem. Until now it was necessary to choose between a well-integrated simple loop optimizer and full polyhedral transformations [12] that translate MLIR dialects into isl-compatible formats. Unfortunately, the conversion costs and community disconnects hindered deeper integration of such approaches into MLIR.

Figure 1. While classical polyhedral compilers have been confined to optional extensions of the compilation flow that interface with external libraries, FPL integrates Presburger arithmetic into MLIR upstream.

With a generalization of schedule trees available in MLIR, the last missing piece is a fast and complete library for Presburger arithmetic. We fill in this gap by contributing a Presburger library, FPL[15], to LLVM. FPL was written from the ground up to meet the community's standards, incentivizing further participation in the development of these mathematical foundations and allowing for the polyhedral compiler to be co-developed with the Presburger library. The Presburger library is already being used in the MLIR Affine dialect for loop fusion[1] and in the CIRCT project for dependence analysis. [2] Thus, we now have a Presburger library co-existing with (a currently somewhat restricted instance of) a polyhedral transformation framework within the LLVM ecosystem, able to make full use of each other.

Having a library deeply integrated with the compiler allows compiler engineers to leverage the full power of Presburger arithmetic while seamlessly translating information

[1] https://github.com/llvm/llvm-project/commit/c8fc5c0385dbb47623c1cca5efa0b96d5e5f8151
[2] https://github.com/llvm/circt/commit/82b9de0f35f3d14a4a6aa69a3842992c7414cb8b

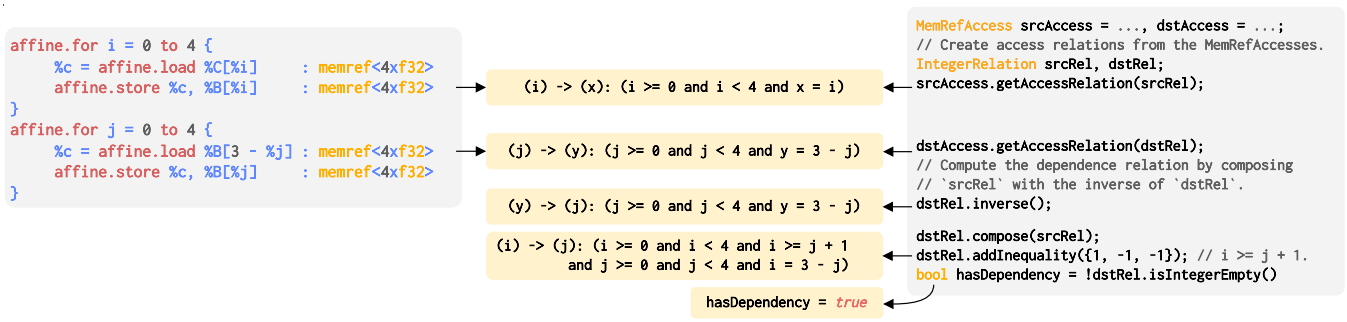

Figure 2. Example flow for performing analysis operations in MLIR with FPL. In this case, we are checking whether it is permissible to fuse the two loops on the left. This is allowed if there is no memory access dependency between an iteration of the i loop with an earlier iteration of the j loop. We can easily check this by converting the `MemRefAccess` objects created from the load/store IR operations to the Presburger library's `IntegerRelations` and performing some Presburger operations.

between the IR level and the mathematical abstractions. This enables closer integration between the IR and the mathematical foundations that would be more difficult to achieve when the library is an external one.

2 Using FPL with MLIR

FPL provides four main data structures for Presburger arithmetic: `IntegerRelation` and `IntegerPolyhedron` are relations and sets respectively defined by a set of affine constraints that must all be satisfied by every element in the set or relation. They support existentially quantified variables, so they can be thought of as projections of convex objects. Both can have symbols, i.e. parameters that have constant but unknown values, such as the size of an array that does not change during the course of a loop but whose size is not known at compile-time. The classes `PresburgerSet` and `PresburgerRelations` are unions of `IntegerPolyhedrons` and `IntegerRelations` respectively – they can support disjunctions of constraints. We support integer-exact set operations like union, intersect, subtract, complement, equality checks, emptiness checks, and lexicographic optimization. For more algorithmic details on these operations, see Pitchanathan et al. [15] or the MLIR documentation.[3]

MLIR provides support to define domain-specific IRs known as dialects. One example is the Affine dialect, which represents affine loop nests in a format suitable for polyhedral compilation. We present an example (Figure 2) of the kind of analyses FPL enables over the Affine dialect. In this example, we wish to check whether the load in the second loop depends on the store of the first one.

In MLIR, a `MemRefAccess` represents a memory load or store access. We take the `MemRefAccesses` built from the store/load instructions and call `getAccessRelation` to obtain an `IntegerRelation` mapping the loop index to the memory location accessed. Inverting the second relation gives us a relation from memory locations accessed by the second loop to its induction variable. Composing these gives a relation representing the dependencies between the iterations i of the first loop to the iterations j of the second. By adding a constraint $i \geq j + 1$, we obtain a relation that has the dependencies from iterations of the first loop to earlier iterations of the second. These two loops can be fused only if there are no such dependencies, which we can check by running the integer emptiness check on this final dependence relation. In this case, there exists a dependence between $j = 1$ and $i = 2$, so fusing the loops would be a miscompile. By providing support for Presburger operations, we lay the foundations for future work leveraging FPL for more complex polyhedral compilation techniques.

[3] https://mlir.llvm.org/doxygen/namespacemlir_1_1presburger.html

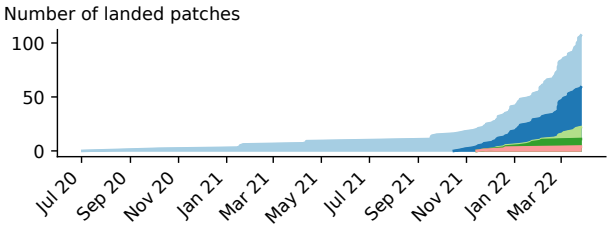

Figure 3. Landed patches over time. Each colour denotes a different contributor; we already have five contributors.

3 Contributing FPL into MLIR

The MLIR Affine dialect initially had some limited support for Presburger-style operations, including a rational emptiness check using Fourier-Motzkin elimination. Over 20 months, 110 patches, and over 1,500 Phabricator comments, we upstreamed support for full Presburger arithmetic to MLIR, including integer-exact operations such as the emptiness check. Apart from the usage of our library in loop fusion, developers outside of our initial team have already expressed interest in working on FPL itself. Over time, more developers have joined our effort (Figure 3). In fact, this nascent component of the LLVM ecosystem already has five contributors.

References

[1] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, 193–205. https://dl.acm.org/doi/10.5555/3314872.3314896

[2] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Sci. Comput. Program.* 72, 1–2 (June 2008), 3–21. https://doi.org/10.1016/j.scico.2007.08.001

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[4] Alexis Fouilhe. 2015. *Revisiting the abstract domain of polyhedra : constraints-only representation and formal proof*. Theses. Université Grenoble Alpes. https://tel.archives-ouvertes.fr/tel-01286086

[5] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly – Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012). https://doi.org/10.1142/S0129626412500107

[6] Tobias Grosser and Torsten Hoefler. 2016. Polly-ACC Transparent Compilation to Heterogeneous Hardware. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/2925426.2926286

[7] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 12 (July 2015), 50 pages. https://doi.org/10.1145/2743016

[8] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A Fast Analytical Model of Fully Associative Caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 816–829. https://doi.org/10.1145/3314221.3314606

[9] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. 1996. The Omega calculator and library, version 1.1. 0. *College Park, MD* 20742 (1996), 18.

[10] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[11] Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra. (1999).

[12] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Affine C in MLIR. (2021). https://acohen.gitlabpages.inria.fr/impact/impact2021/papers/IMPACT_2021_paper_1.pdf Not a formal proceedings.

[13] Kedar S. Namjoshi and Nimit Singhania. 2016. Loopy: Programmable and Formally Verified Loop Transformations. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science)*, Xavier Rival (Ed.), Vol. 9837. Springer, 383–402. https://doi.org/10.1007/978-3-662-53413-7_19

[14] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. 2020. Automated Derivation of Parametric Data Movement Lower Bounds for Affine Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 808–822. https://doi.org/10.1145/3385412.3385989

[15] Arjun Pitchanathan, Christian Ulmann, Michel Weber, Torsten Hoefler, and Tobias Grosser. 2021. FPL: Fast Presburger Arithmetic through Transprecision. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 162 (oct 2021), 26 pages. https://doi.org/10.1145/3485539

[16] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-Based Data Reuse Optimization for Configurable Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 29–38. https://doi.org/10.1145/2435264.2435273

[17] Konrad Trifunović, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. 2010. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *2nd GCC Research Opportunities Workshop (GROW)*. http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.220.3386

[18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv e-prints*, Article arXiv:1802.04730 (Feb. 2018), arXiv:1802.04730 pages. arXiv:cs.PL/1802.04730

[19] Sven Verdoolaege. 2016. Presburger formulas and polyhedral compilation. (2016).

[20] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule trees. In *International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria.*