Building a Static HLS Pass with FPL

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

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

Julian Oppermann
TU Darmstadt
oppermann@esa.tu-darmstadt.de

Mike Urbach
SiFive
mike.urbach@sifive.com

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

Abstract

Compiler infrastructure like LLVM/MLIR provides modular and extensible building blocks for reuse in compiler development. The previously existing polyhedral building blocks depended on external solvers. Due to the perceived cost of depending on such external tools, these building blocks were used only in fully polyhedral pipelines. For hybrid or one-off use cases, selective reimplementation has been preferred over taking the dependency; several reimplementations of sub-polyhedral solvers now exist in the ecosystem. With the introduction of a fast Presburger library (FPL) in MLIR, many core polyhedral building blocks are now available in-tree. As a result, it is now possible to use polyhedral techniques within the MLIR-based CIRCT project. Using the development of a static high-level synthesis (HLS) pass in CIRCT as a case study, we show the impact of FPL's availability upstream.

1 Introduction

Traditional full-scale polyhedral loop optimization depends on solving large-scale integer linear programming (ILP) problems. While this has been a successful path, there is a tendency for polyhedral compilation to be seen as an all or nothing approach that necessarily involves polyhedral loop scheduling. Recent work in polyhedral compilation has explored mixing AST-based techniques with polyhedral techniques [4, 6, 11]. Rather than deferring to the polyhedral model for the entire optimization pipeline, the Presburger solver is used to compute specific properties of the code in a way that scales better to larger programs; two examples are static analysis [7] and cache performance modelling [5].

Polyhedral compilation in the LLVM/MLIR ecosystem has traditionally depended on external solvers, but there is a high perceived cost of taking on this dependency. As a result, only full-scale polyhedral loop optimizers have used these external tools. For more minor one-off use cases, the community has reimplemented parts of solvers in-tree and has done so several times in different modules of LLVM. However, implementing a full Presburger library involves a significant amount of effort, so these applications have made do with partial reimplementations that support only sub-polyhedral queries.

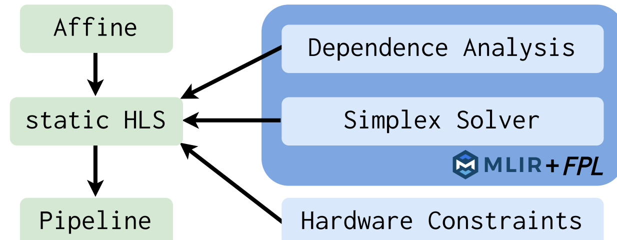

Figure 1. FPL faciliates the design of a static HLS pass for MLIR by facilitating the computation of memory dependences and solving optimization problems as they arise when mapping loops to a hardware pipeline.

The upstreaming of FPL introduced, for the first time, a full Presburger solver in LLVM/MLIR. We show the impact of FPL with a case study from the MLIR-based CIRCT project by describing how a critical part of high-level synthesis flows leverages FPL in multiple ways.

2 The MLIR compiler framework

MLIR [6] is a compiler framework in the LLVM ecosystem that provides building blocks to create, analyze, and transform domain-specific intermediate representations (IRs), thus facilitating the development of domain-specific compilers.

The Affine dialect, representing affine loops, is one of the core dialects in MLIR and represents static control parts. Listing 1 shows an example of a program in the Affine dialect. MLIR also provides an in-tree Presburger library, FPL [9], and several utilities for interaction between FPL and the Affine dialect for mathematical analysis.

The CIRCT project is a collection of domain-specific IRs and compiler passes focused on hardware design. It is built on the MLIR framework and supports several hardware compiler flows, including multiple forms of high-level synthesis (HLS). HLS transforms an untimed, higher-level hardware description into a cycle-level description. Notably, CIRCT provides the *Pipeline* dialect, which captures timed hardware pipeline descriptions.

Our case study follows the development of a static HLS flow that transforms a program in the Affine dialect to CIRCT's Pipeline dialect, which is then further lowered into CIRCT's

```
affine.for %i = 2 to 64 iter_args(%arg3 = 0) {
  S0: %1 = affine.load %mem[%i]
  S1: %3 = arith.muli %1, %arg3
  S2: %2 = affine.load %mem[%i - 2]
  S3: %4 = arith.addi %3, %2
  S4: affine.store %4, %mem[%i]
  S5: affine.yield %4
}
```

Listing 1. An example program in the Affine dialect.

low-level dialects. This flow applies polyhedral analysis in a modular and domain-specific way, to perform dependence analysis and scheduling. Since CIRCT is MLIR-based, it can leverage FPL for this analysis.

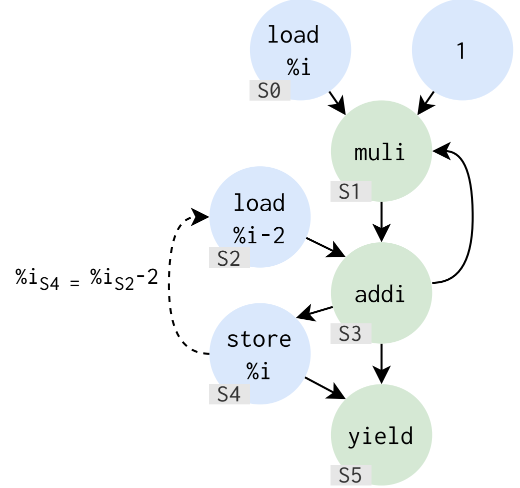

Figure 2. The cyclic scheduling problem corresponding to Listing 1, assuming a latency of 1 for load/store operations, 3 for the multiplication, and 0 for the addition. The arrows represent the dependences between operations. The dashed arrows represent memory dependences. The backedge $S3$ to $S1$ has a distance of 1, meaning the multiplication depends on the result of the previous's iteration's addition. Analogously, the memory dependence $S4$ to $S2$ has a distance of 2.

3 Lowering Affine to Pipeline

We perform HLS for Affine programs by lowering a high-level program to the Affine dialect via an MLIR frontend and then using CIRCT's AffineToPipeline pass [12] to further lower this into the Pipeline dialect, which is then converted to lower level CIRCT dialects. The AffineToPipeline pass constitutes a critical part of an HLS flow that takes high-level machine learning programs in TensorFlow [1] or PyTorch [8] down to low-level hardware descriptions in System Verilog. We use the development of this pass as a case study to examine how FPL enables the use of polyhedral techniques in a production-focused compiler like CIRCT.

The AfffineToPipeline pass converts a sequentially described program in the Affine dialect to a statically scheduled pipelined program. This requires scheduling the operations to a pipeline in such a way that the program semantics are preserved. To ensure this, it computes dependences between operations in the program, which constrain the relative orderings of these operations. For example, in Listing 1, $S2$ depends on $S4$, as shown in Figure 2. We use polyhedral memory dependence analysis to detect such dependences.

Finally, the pass finds an optimal schedule of the operations and outputs a statically scheduled pipeline. Only valid schedules are considered; valid schedules are those that respect the dependences as well as hardware constraints. Figure 3 shows the pipeline generated by the running example.

3.1 Dependence Analysis

The lowering must preserve both *structural* and *memory* dependences. Structural dependences are those due to explicit data flow via register dependences in the program. For example, in Listing 1, $S3$ has a structural dependency on $S1$ and $S2$, since it uses the results of those statements. Memory dependences, on the other hand, occur when an operation reads some memory written by another operation. For example, in Listing 1 the value written by $S4$ is read by $S2$ in a future iteration of the loop.

While structural dependences are immediately clear from the IR, memory dependence analysis is more involved. To obtain precise results, polyhedral dependence analysis is required. We use the implementation provided by MLIR, which is based on FPL. The Affine dialect assumes sequential execution of statements, whereas we also need to consider statements possibly being reodered. Therefore, we extend the dependences to take this into account.

3.2 Scheduling

CIRCT's static scheduling infrastructure [2] provides an extensible scheduling model and a growing library of scheduling algorithms for typical operator scheduling applications. CIRCT provides multiple scheduling models for different use-cases. In this paper we will focus on the CyclicProblem model, which considers constraints related to the pipelined execution of loops in the absence of resource constraints. The underlying assumption is that every iteration begins a fixed number of time steps after the previous iteration was started. This interval is called the *initiation interval (II)*. In order to maximize performance, we find the smallest valid II.

The model assigns a fixed execution time for each statement in the loop body as an offset relative to the start of the iteration. An II is valid iff we can assign these start times such that for every dependency $S \to T$ between two statements, T starts executing after S completes. This can be written as

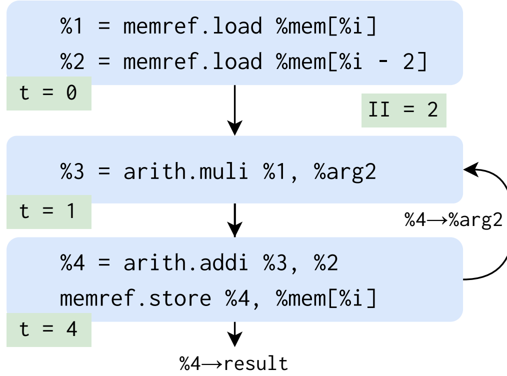

Figure 3. A scheduled pipeline output by the AffineToPipeline pass.

a linear constraint

$$\text{startTime}_S + \text{latency}_S \leq \text{startTime}_T + \text{depDist}(S \to T) \cdot II$$

where latency_S is the latency of the instruction executed at S, and $\text{depDist}(S \to T)$ is the dependence distance between S and T. For example, in Figure 2, the accessed memory locations cause $S4$ to depend on the completion of the $S2$ instance two iterations earlier. Note that the start times and II have to be integers, and the constraints and objective are linear, so we can find the optimal II by integer linear programming. Figure 3 shows a pipeline generated for the running example using this framework.

The current reference scheduler for this model implements the approach proposed by de Dinechin [3]. This scheduler uses a tailored but unoptimized implementation of the parametric dual simplex algorithm. We added a new implementation for scheduling using the simplex solver available in FPL. This implementation is currently being upstreamed to CIRCT[1].

Our implementation uses FPL's rational lexmin solver. We encode the problem variables as a vector: (II, objectives, startTimes) and find the lexicographically minimum vector that satisfies the problem constraints. Due to how the variables are ordered, the II is minimized first, and then the objectives. If the II is non-integral in the result, we add a constraint on the II to be the ceiling of the solved value and solve again. The start times of the operations are guaranteed to be integer from how the scheduling model creates the constraints.

There are several benefits to the newer implementation. Since FPL is available in MLIR, there is no extra cost of external dependences. Also, there are plans to vectorize FPL upstream, which would improve our performance [10].

FPL has support for more complex solvers, including integer linear programming and parameteric integer linear programming, which are currently only available via an external library dependency to the scheduler implementations in CIRCT. Thus, the porting to FPL opens up several possibilities for the scheduling framework, ranging from more complex objectives to better analysis without having an external dependency. For example, the problem constraints can be relaxed to allow for rational solutions and solved using the ILP solver. The problems can also be posed parametrically to obtain a solution parameteric in the start time of a subset of operations using the Parameteric ILP solver. For example, solving the problem in Figure 2 with the start time of $S1$ as a parameter allows us to explore the solution space, parameterized by the start time of $S1$.

[1] https://github.com/llvm/circt/pull/4517

4 Conclusion

FPL enables the use of polyhedral techniques in the production-quality LLVM/MLIR compiler framework. This makes polyhedral compilation more accessible to the broader LLVM/MLIR community, as we saw in the case study of a static HLS pass for CIRCT. The availability of FPL upstream makes it easier to experiment with smaller and more targeted uses of polyhedral techniques.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

[2] CIRCT. [n. d.]. Static scheduling infrastructure. ([n. d.]). https://circt.llvm.org/docs/Scheduling/.

[3] Benoît Dupont de Dinechin. 1994. Simplex Scheduling: More than Lifetime-Sensitive Instruction Scheduling. *PRISM 1994.22* (1994).

[4] Roman Gareev, Tobias Grosser, and Michael Kruse. 2018. High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach. *ACM Trans. Archit. Code Optim.* 15, 3, Article 34 (sep 2018), 27 pages. https://doi.org/10.1145/3235029

[5] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2020. A Fast Analytical Model of Fully Associative Caches. *arXiv e-prints*, Article arXiv:2001.01653 (Jan. 2020), arXiv:2001.01653 pages. arXiv:cs.PF/2001.01653

[6] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[7] 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

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[9] Arjun Pitchanathan, Kunwar Shaanjeet Singh Grover, Michel Weber, and Tobias Grosser. 2022. Bringing Presburger Arithmetic to MLIR with FPL. In *Proceedings of the 12th International Workshop on Polyhedral Compilation Techniques*. Budapest, Hungary.

[10] 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

[11] Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. 2014. Oil and Water Can Mix: An Integration of Polyhedral and AST-Based Transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, 287–298. https://doi.org/10.1109/SC.2014.29

[12] Mike Urbach and Morten Borup Petersen. 2022. HLS from PyTorch to System Verilog with MLIR and CIRCT. *latte'22* (2022).