Efficiently Learning Locality Optimizations by Decomposing Transformation Domains

THARINDU PATABANDI and MARY HALL*, University of Utah, USA

ACM Reference Format:
Tharindu Patabandi and Mary Hall. 2022. Efficiently Learning Locality Optimizations by Decomposing Transformation Domains. 1, 1 (September 2022), 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Optimizing compilers for efficient machine learning are more important now than ever due to the rising ubiquity of machine learning in all facets of life imaginable. They are essential tools for achieving architecture-specific high-performance code. Compilers exploit various techniques to modify the runtime behavior of input programs. *Loop transformations* alter the execution order of the iteration space of a loop nest and generate functionally equivalent code variants with better target resource utilization. As a result, compilers achieve better locality exploitation and improved parallelism, both desirable properties towards high-performance.

However, deriving effective transformation schedules is non-trivial and depends heavily on the intricate interactions between the computation and the target architecture. An instance of this behavior is demonstrated in Figure 1 in which we plot the performance distribution of the varying loop tile and loop permutation combinations for three convolutional layers. It is evident from these figures that sub-optimal transformation choices can substantially deteriorate the overall performance of the resulting loop nest.

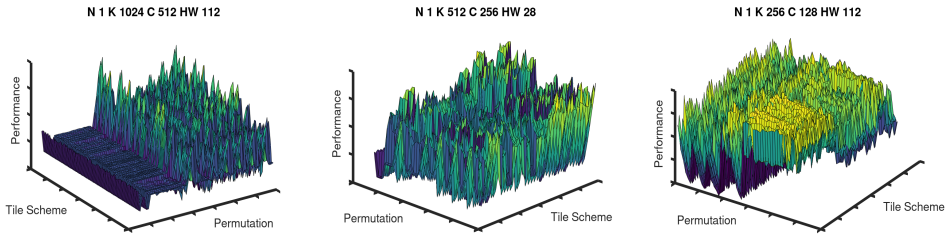

Fig. 1. Performance distribution of various tile-permutation combinations

For decades, Machine Learning (ML) has provided a promising platform for learned compiler optimizations [Moss et al. 1997], due to the relative ease of deploying learned models to derive an effective sequence of loop transformations. In recent years, ML driven compiler optimization and program synthesis has achieved widespread popularity, as a result of the advent of powerful machine learning abstractions and the omnipresence of their efficient implementations. Also, the relative simplicity of the overall development and deployment processes compared to alternatives such as empirical autotuning and analytical models is noteworthy. This recent success has motivated the exploration of ML in the context of compilers, programming languages, and synthesis a dedicated

*(advisor)

Authors' address: Tharindu Patabandi, tharindu@cs.utah.edu; Mary Hall, mhall@cs.utah.edu, University of Utah, Salt Lake City, Utah, USA, 84112.

research discipline under the banner of *Machine Programming*, a term coined by Gottschlich et al. [2018].

Loop optimization is often framed as *Design Space Exploration* (DSE) in which each point represents a unique parameter combination of a schedule, and entails a form of search over the schedule space either exhaustively or with pruning. This process involves iterating over a huge combinatorial space of schedule instances, generating the corresponding code, and measuring performance metrics of the code variants (iterative compilation). As a result, many studies only focus on a handful of values for each transformation. Even ML inspired solutions are not exempt from this complexity as the aforementioned exploration is necessary for building training sets.

However, we identify the inherent search complexity is rooted in what we refer to as *Multiplicative Domain Formulation(MDF)*. Say, we have N transformations with their respective schedule domains $S_1, S_2, ..., S_N$. With MDF, the exploration space consists of $O(|S_1| \times |S_2| \times ... \times |S_N|)$ code variants. We argue this combinatorial complexity can be alleviated by a domain approximation that we refer to as *Additive Domain Formulation (ADF)*. In ADF, the search space consists of $O(|S_1| + |S_2| + ... + |S_N|)$ code variants. With ADF, the challenge is to derive a profitable sequence of transformations by using a collection of *singular model*s, a model that specializes in its respective transformation domain S_i. Compared to the exponentially large *MDF*, this is an order of magnitude reduction in the number of code variants required to be measured in the generation of training data. This work makes the following contributions.

- Two singular models for predicting high-performance tile schemes and loop permutations to derive data locality optimizations for the Conv2d operator that improve baseline performance up to 6.6×
- A novel design space approximation called *Additive Domain Formulation (ADF)*, and an ADF-inspired inference setting, *Composed Singular Prediction (CSP)*, for navigating the search space that significantly reduces the training data generation overhead of the tile model
- An MLIR-powered code generator and the related compiler infrastructure to utilize CSP inference, and generate tiled, unrolled, and vectorized AVX-512 code for Intel Xeon architectures
- A performance evaluation of the optimized Conv2d kernel on Intel Xeon CascadeLake platform that achieves an average 1.4× (max 4.0×) speedup over the state-of-the-art library, Intel oneDNN while saving ~105.3× in training data collection time

REFERENCES

Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B Tenenbaum, and Tim Mattson. 2018. The three pillars of machine programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 69–80.

J. Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanovic, Carla Brodley, and David Scheeff. 1997. Learning to Schedule Straight-Line Code. In *Advances in Neural Information Processing Systems*, M. Jordan, M. Kearns, and S. Solla (Eds.), Vol. 10. MIT Press. https://proceedings.neurips.cc/paper/1997/file/bcc0d400288793e8bdcd7c19a8ac0c2b-Paper.pdf