Abstract

Automating locality optimization is still an open problem for compiler writers. Compiler-based approaches, guided by analytical cost models, have achieved some success in matching high performance libraries on a restricted set of computations such as general matrix multiply (GEMM). On the other hand, library-based approaches may present some open scalability concerns. Recent developments in convolutional neural networks has seen an explosion of models, each with differing combinations of parameters. Manually tuning each of these configurations can take many development months. Further, these operations are called multiple times during machine learning training, which necessitates highly optimized implementations.

2D convolutional operators are unique in that they consist of 7-deep loop nests with different loops carrying reuse for different tensors, making the problem of identifying an optimal loop ordering hard. We devise a machine learning-based compiler which learns a regression model, correlating performance with the loop order. We integrate this model with other traditional compiler analysis for transformations such as loop unrolling and vectorization, relying on the Multi-Level Intermediate Representation (MLIR) compiler framework. We achieve an average speedup of 1.67× and 1.41× against oneDNN for 2D convolution forward and weight update kernels respectively. We are also at 0.88× and 0.96× the performance of oneDNN’s best performing implementation which applies additional data layout transformations.

**CCS Concepts:** • Software and its engineering → Source code generation; • Computing methodologies → Neural networks.

**Keywords:** Compilers, loop transformations, convolutional neural networks, machine learning

**ACM Reference Format:**

1 Introduction

Locality optimizations have complex interactions, and target limited storage for which significant differences in performance can result from subtle differences in the optimization strategy. Recent research has shown that the best sequence of transformations to apply might be specific to the function itself [3, 4, 16]. However, compiler writers trade-off performance for generality. Consequently, there is no one-size-fits-all compiler framework. On the other hand, manually implementing optimizations to achieve ninja-level performance can be a time consuming process; it does not scale. The problem is exacerbated by the growing heterogeneity of architectural platforms, which themselves are evolving.

Compiler-based approaches [22, 25] that are guided by analytical cost models have achieved some success in mimicking ninja optimizations for a restricted set of computational kernels, of which the generalized matrix multiply kernel is the prime exemplar. In recent years, attempts have been made to extend analytical models to higher-order tensor applications such as the convolution operator [18, 20]. There has also been a steady increase in the number of domain specific compilers such as Halide [21], Spiral [19], FFTW [10], and
Pochoir [24], indicating that it might be profitable to design compilers specialized to the function. In other words, the belief is that many codes with a similar semantic structure can be optimized in a closely related fashion.

This project aims to use machine learning (ML) to detect codes with such semantic similarities so that existing compiler heuristics can be reused to optimize them, thereby improving software performance with a learned model, rather than an analytical one that must be designed and implemented by a compiler developer. ML-based approaches [2, 13] have demonstrated state-of-the-art and in some cases better performance than those based on analytical modeling. From the view of Gottschlich et al.’s three pillars, this work aims to understand the intention (i.e., semantics) of compiler heuristics and then adapt them for further optimization refinement [12].

While we espouse an ML-based approach to efficiently mine the space of possible optimizations, we recognize that this space of optimizations is extremely large. Using ML to predict the optimal transformation sequence, as demonstrated in recent approaches, tends to require sophisticated search approaches and large training times. We reduce the complexity of this space by focusing on loop permutation, which establishes the order of the loops in the nest. Learning just this optimization makes training time efficient. We utilize well-studied compiler analysis techniques for the other transformations that follow permutation. Further, we utilize compiler analysis to extract meta-features such as reuse information to aid the machine learning model to predict optimizations with pertinent features.

This work makes the following contributions:

- A domain-specific compiler for 2D convolution codes that combines analytical and machine learning models within a unified optimization framework.
- A regression-based machine learning model that accurately predicts the performance of code variants with different loop permutations by utilizing code features including loop bounds, array accesses, reuse and dependence information.
- A compiler and code generation system implemented using the Multi-Level Intermediate Representation (MLIR) [17] framework that achieves performance that is competitive with the state-of-the-art hand-optimized library, oneDNN [1], on multi-core CPU platforms. We achieve an average speedup of 1.67× and 1.41× for 2D convolution forward and weight update kernels respectively. We are also at 0.88× and 0.96× the performance of oneDNN’s best performing implementation which applies additional data layout transformations.

2 Motivation

2D convolutional operators are the workhorses of modern image processing and machine learning applications. They routinely appear in convolutional neural networks such as ResNet, GoogleNet, Yolo and VGG, to name a few. Typically, they are instantiated with many different combinations of image height and width, filter height and width, as well as the input and output channel dimensions. For example GoogleNet-v1 has more than 150 such configurations. These kernels are then called multiple times during training, hence necessitating highly tuned and optimized implementations of such operators to keep the running time tractable.

The forward, backward and weight update kernels for the 2D convolutional operator consist of a 7-dimensional loop nest and 3 array references. Optimizing each of these kernels is not straightforward as differing loop dimensions carry reuse of different arrays. Furthermore, loops that carry reuse for one reference might not have reuse for other arrays. As a result, the optimization of these kernels tends to be input-dependent and predicting optimal loop orders analytically is difficult.

Figure 1 serves to illustrate the importance of loop permutation for the 2D convolution operator. It highlights the variation in performance, just by changing the loop order alone and keeping other applied optimizations invariant. As observed from Figure 1 there is a difference of more than an order of magnitude between the worst and best performing code variants. We advocate a hybrid approach: (i) utilize machine learning to mine the huge combinatorial search...
space of possible loop permutations to learn an optimal loop permutation given the input loop dimensions, array sizes and reuse patterns, and (ii) apply well-known analyses and heuristics for optimizations such as unrolling and vectorization.

Our proposed ML-based compiler framework is shown in Figure 4. The traditional components of the compiler framework are highlighted in blue, while the ML components are in red. The traditional components of the compiler are namely, the parsing, dependence extraction, intermediate representation analysis, transformation, and code generation. The ML components are the search space generation, feature encoding, and the specific model architecture used during training and inference.

3 Methodology

Figure 3 outlines some of the analysis and pre-processing done by the compiler prior to code generation such as the initial representation of the loop nest, the annotations that the compiler adds to it in stages, and the feature extraction from the compiler’s intermediate representation to the data set that is subsequently used during the training phase. Dependence analysis and the compiler’s profitability analysis to identify vector loops and unroll loops as described in Section 3.4 marks loops for vectorization (colored in blue) and unrolling (green).

Following the compiler analysis, an analysis of the search space of possible loop orderings potentially results in pruning. This is achieved by fixing the positions of certain loops based on heuristics. In Figure 3, the loops to be placed in

---

**Figure 2.** 2D Convolution benchmarks in MLIR

**Figure 3.** ML4COpt: Analysis and dataset generation.

**Figure 4.** ML4COpt: Machine learning-based compiler system.
the outermost (highlighted in red) and innermost (colored black) positions are chosen to restrict the search space of possible permutations. Finally, the annotated loop order is passed to the code generator, so that the code is transformed accordingly.

3.1 Search Space Enumeration and Training Data
For the 2D convolutional computations, we limit the search space of possible loop orders. All three convolutional computations (FP, BP, and WU) consist of a 7-dimensional loop nest, which corresponds to 5040 legal loop permutations, keeping the loop bound values invariant. In order to limit the search space complexity, we fix the positions of the innermost and outermost loops based on analytical heuristics. For example the mini-batch loop (N) is typically placed outermost in most cases for 2D convolutional computations, while the inner loop is chosen to maximize register reuse and as a result, to minimize loads and stores in the innermost loop. This allows us to reduce the search space size from 5040 ($7!$) to 120 ($5!$).

We sample 128 different configurations of loop bounds for each of the forward, backward and weight update convolution kernels. The range of the loop bounds was chosen to reflect those that are commonly encountered in CNN benchmarks. Specifically the ranges of the loop bounds corresponding to the height and width of the images was chosen in the range [7, 224] while the square filter dimensions were fixed with the value 3. The mini-batch dimensions were varied between 64 and 128. Finally the input and output channel dimensions were chosen from the range [16, 512]. For each configuration of loop bounds all 120 loop permutations were exhaustively explored with their respective performance recorded in GFLOPS/s.

3.2 Features
We extract a set of static features from the MLIR source representation in order to train the model. These features are representative of loop bounds and types of data dependences present between different memory accesses in the program. We omit control dependences in this study because we observe uniform control flow across all target benchmarks.

Therefore, we hypothesize that data dependence structure is more important in making the loop permutation decision.

The static features extracted from the code’s MLIR is shown in Figure 5. A, B, C are the three 4-D tensors involved in the computation. N, C, K, H, W, R, S represent standard convolution parameters.

The set of loop bounds of the 7-dimensional loop nest is the first portion of the feature representation. Loop bounds are important towards determining an optimal loop permutation because varying loop bounds can help induce different memory/cache footprints and subsequently, cache misses.

The latter portion of the feature vector is allocated for dependence-based data reuse information. The reuse information is based on the compiler’s analysis of the data dependences in each computation and a tagging of each loop for the type of reuse it carries for each array in the computation. $d^X_i$ is the type of reuse carried by loop $i$ with respect to tensor $X$. We currently classify the reuse corresponding to 5 types: spatial, temporal, partial, stencil and none. Namely, spatial reuse refers to loops for which distinct iterations reference the same cache line, temporal reuse is present in loops for which distinct iterations reference the same word, and stencil refers to a reuse pattern commonly observed in stencil computations where two or more ‘coupled’ loop indices carry reuse while partial reuse refers to the scenario where a loop that has spatial locality has been strip-mined and vectorized, but has additional reuse due to the outer loop that remains after strip-mining and vectorization and none indicates no reuse.

Given a feature vector, we would like to know the predicted performance of the given code variant. However, for the same bounds and the same dependence structure, this performance is dependent on the input loop permutation of the nest. Therefore, a feature vector should reflect this permutation so the model can differentiate among the loop nests with the same loop bounds but different loop permutation. To this end, we transform the feature vector with a permutation-sensitive positional encoding. This transformation is shown in figure 5. Each loop bound $i$ is projected onto...
were trained with 2000 epochs. We will demonstrate the work architecture but with loop bounds

\[ \forall j \in \{0, \ldots, 6\}, \quad v_i[j] = \begin{cases} b_i & \text{if } \text{pos}(i) == j \\ 0 & \text{otherwise} \end{cases} \]

where pos(i) is the position of the loop i. This positional transformation maps the bounds information to a 7 × 7 permutation space. With this encoding, we are able to identify the exact permutation of a loop nest. The same encoding is applied to the dependence features in order to reflect the fact that each reuse permutation is associated with a particular loop level. The transformation results in a final feature vector with (7 × 7) + (3 × 7 × 7) = 196 positional features.

### 3.3 Supervised Learning

To establish a correlation between the features and the measured empirical performance (in GFLOPS/s), we employ a fully connected deep regression model with mean-squared error (MSE) as the loss function. We train 4 variants of this model, one for each benchmark (i.e., Conv-FP, Conv-BP, and Conv-WU), and a last model that uses data from all three benchmarks. The model consists of 5 fully connected layers of sizes [(196, 60), (60, 30), (30, 15), (15, 7), (7, 1)] and rectified linear units (ReLU) as activation.

A second set of models were trained with the same network architecture but with loop bounds as the only feature. The purpose of these models was to observe if the performance can be predicted with less features or identify potential redundancies in the feature vector. We refer to this model as Bounds- or B-model and the previous model that uses both bounds and dependence information Bounds-Dependence or BD-model. Training used learning rate \( l_r = 10^{-5} \) with Adaptive Moment Estimation (Adam) as the optimizer. All models were trained with 2000 epochs. We will demonstrate the inference accuracy of these models in Section 4.

### 3.4 Analytical Model

Compiler analysis determines the loop(s) to vectorize, unroll and their associated unroll factors. Firstly, dependence analysis is carried out by the compiler to identify loops that do not carry dependences. These loops are candidates for vectorization. A subsequent analysis of the data layout of the input tensors determines which of these candidate loops have additional spatial locality, for example, if they are the fastest changing dimension of the tensor. The loop with the maximum spatial locality is picked for vectorization. This loop is strip-mined by the SIMD width.

Similar to the vectorization analysis, the unroll loops are again determined by dependence analysis for loops that do not carry output dependences. We limit the number of unrolled loops to two, and search over combinations of unroll factors. For each combination of unroll loops and factors, we evaluate the register footprint by examining the reuse of the input tensors for the unrolled loops. Unless the loop chosen for unrolling has temporal reuse for the tensor, the register footprint would increase with a non-unit unroll factor. We then compute the effective arithmetic intensity of the unroll loop configuration by measuring the ratio of floating point operations to the register footprint. Finally the configuration with the maximum arithmetic intensity, subject to the additional constraint that the register footprint is less than or equal to the maximum vector register file size, is fixed for subsequent feature encoding and code generation.

### 4 Evaluation

We evaluate the supervised learning approach outlined in Section 3.3 to predict the performance of 2D convolution operators (FP, BP and WU) on different layer configurations. We ran our experiments on an Intel(R) Xeon(R) Platinum 8180 CPU, with a clock frequency of 2.50GHz and a maximum turbo frequency of 3.8 GHz. The dual socket processor has 28 cores per socket. The system has 32KB of L1 cache, 1MB of L2 cache and 38.5 MB of L3 cache.

Once trained, the model predicts the performance of a given loop permutation of candidate configurations. A set of configurations ranked by the predicted performance are then input to the code generator.

**Figure 6.** Inference accuracy for varying error tolerance levels. Bar plots represent the Bounds (B-) models and line plots represent Bounds-Dependence (BD-) models.

Our implementation of the 2D convolution operators is based on TensorFlow’s NHWC data layout for the input and output tensor, and RSCK layout for the weight tensor, for the forward and weight update kernels. For the backward

---

1 GPUs are crucial in deploying CNN workloads. However, heterogeneous device mapping is beyond the scope of this research and, as such, we did not evaluate the model on GPUs. However, the authors hypothesize that provided the platform specific heuristics and the infrastructure to seamlessly generate GPU code, the feature representation and the FCNN model can be retrained across platforms to achieve similar results.
we use a custom RSCK layout for the weight tensor. We evaluate the performance of our single-threaded implementations as MLIR’s support for multi-threading is not fully functional at this time.

4.1 Inference Accuracy

Each model was trained on a random 80% split of the dataset with the remaining 20% forming the test set. We did not perform cross validation during training. The model predicts the performance for each of the 120 permutations in the test set, with none of the permutations of test layers being present during training.

We report the performance of the ML model in terms of (i) The accuracy of predicted results relative to the actual performance and (ii) The absolute performance in GFLOPS/s of the predicted optimal permutations in comparison to the empirical best performing code variants.

In Figure 6, we demonstrate the percentage accuracy of both B- and BD- models for varying degrees of error tolerance $\epsilon$. If $|\text{model}(x) - \hat{y}| \leq \epsilon$ for $\epsilon \in \mathbb{R}^+$, we say the prediction for input $x$ is correct within error tolerance $\pm \epsilon$ GFLOPS/s, where $\hat{y} = \text{label}(x)$. We observe that for each $\epsilon$, BD- models outperform their corresponding B- model’s accuracy. This proves that data dependence and reuse information in the feature vector enable learning a more informed correlation between the input domain and performance. Therefore, we continue to use BD- models for the rest of the evaluation.

Figure 7 reports the absolute performance in GFLOPS/s of the best code variant as predicted by the model, relative to the actual best-performing code variant. The layer configurations used in the test sets are summarized in Table 1. On average the performance of the best code variant as predicted by the model is 95%, 94% and 95% of the best performing code variant obtained by exhaustive search for forward, backward, and weight update respectively. This indicates that even when the model mispredicts the best performing configuration, the performance of the predicted configuration is still in the vicinity of the best performing configuration. In our training data, barring configuration C19 for backward, the performance of the predicted best performing code variant correlates strongly with the absolute best performing variant. In the case of C19, the loop bounds encountered are at the extremal values of the range we set for our experiments, indicating that we need more training data around these values for greater accuracy for these configurations.

The absolute performance graph also elucidates certain aspects of our analytical model. The analytical model’s heuristics for unroll loops, factors, and vectorized loop do not seem to negatively impact the performance of the best performing code variants. ML could further improve the performance for these configurations if we trained with more combinations of unrolled loops and factors. Currently we only experiment with one predefined set of unrolled loop configurations and exposing it to the ML model is future work.

4.2 Comparison to oneDNN Library

In this experiment, we evaluate the set of 20 2D convolution layers that we predicted during inference by comparing their performance with that of oneDNN-v2.2 [1]. We compare the performance of the 2D Conv-FP kernel in Figure 8 (top) and Conv-WU in Figure 8 (bottom) against two different data layout configurations of oneDNN. Note that the system peak is approximately 243 GFLOPs on this machine.

In our implementation (coded ml4copt in Figure 8), we adopt NHWC/RSCK layout similar to oneDNN(nhwc+rsck). We also compare with the best performing implementation within oneDNN which applies data layout transformations to the weight tensor to convert it from the default RSCK data layout to a blocked weight layout. For Conv-FP, we outperform oneDNN(nhwc+rsck) on 19 out of 20 layers and, 17 of 20 for Conv-WU. We achieve an average speedup of 1.67× and 1.41× for Conv-FP and Conv-WU respectively. We are also at 0.88× and 0.96× the performance of oneDNN’s best performing implementation which applies additional data layout transformations.
Table 1. FP, BP, and WU inference layers used in model testing. $R \times S = 3 \times 3$, $stride = 1$

<table>
<thead>
<tr>
<th>N</th>
<th>C</th>
<th>K</th>
<th>HW</th>
<th>N</th>
<th>C</th>
<th>K</th>
<th>HW</th>
<th>N</th>
<th>C</th>
<th>K</th>
<th>HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>64</td>
<td>64</td>
<td>128</td>
<td>14</td>
<td>64</td>
<td>256</td>
<td>28</td>
<td>64</td>
<td>512</td>
<td>256</td>
<td>112</td>
</tr>
<tr>
<td>C2</td>
<td>128</td>
<td>64</td>
<td>512</td>
<td>14</td>
<td>64</td>
<td>256</td>
<td>28</td>
<td>64</td>
<td>512</td>
<td>256</td>
<td>28</td>
</tr>
<tr>
<td>C3</td>
<td>128</td>
<td>128</td>
<td>256</td>
<td>28</td>
<td>128</td>
<td>256</td>
<td>28</td>
<td>128</td>
<td>512</td>
<td>256</td>
<td>112</td>
</tr>
<tr>
<td>C4</td>
<td>128</td>
<td>64</td>
<td>512</td>
<td>56</td>
<td>64</td>
<td>128</td>
<td>512</td>
<td>28</td>
<td>64</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>C5</td>
<td>128</td>
<td>64</td>
<td>256</td>
<td>112</td>
<td>64</td>
<td>64</td>
<td>128</td>
<td>28</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>C6</td>
<td>128</td>
<td>64</td>
<td>256</td>
<td>14</td>
<td>64</td>
<td>128</td>
<td>512</td>
<td>28</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>C7</td>
<td>128</td>
<td>64</td>
<td>128</td>
<td>512</td>
<td>64</td>
<td>256</td>
<td>128</td>
<td>28</td>
<td>128</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>C8</td>
<td>128</td>
<td>64</td>
<td>128</td>
<td>512</td>
<td>128</td>
<td>256</td>
<td>128</td>
<td>28</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>C9</td>
<td>128</td>
<td>64</td>
<td>128</td>
<td>512</td>
<td>64</td>
<td>256</td>
<td>128</td>
<td>28</td>
<td>128</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>C10</td>
<td>128</td>
<td>128</td>
<td>256</td>
<td>56</td>
<td>128</td>
<td>256</td>
<td>256</td>
<td>28</td>
<td>128</td>
<td>256</td>
<td>256</td>
</tr>
</tbody>
</table>

Figure 8. Performance comparison of the best predicted performance for the layers in the test sets with Intel OneDNN. Vertical axis is performance in GFLOPs/s. (Top) Conv-FP, (Bottom) Conv-WU

In summary, the model is able to achieve $\sim 90\%$ of oneDNN’s best performing implementations via an automated compiler-based approach, demonstrating the performance impact of finding an optimizing loop permutation towards overall performance.

4.3 Permutation Analysis

In this section, we aim to analyze the optimal permutations for the convolutional layers in the test sets.

Figure 9 compares the performance of the predictions made for the Conv-FP test layers for each of the 120 permutations against their respective labels. Green regions show good performance whereas red regions represent low performing permutations. We observe in the label distribution (right) a clear distinction between good and bad permutation clusters. These clusters are not necessarily consistent across different applications and/or platforms due to the interaction between different data reuse patterns and cache/memory structures. Therefore, re-training the model will be required to achieve similar accuracy and performance for different applications and/or hardware platforms. However, given the infrastructure developed for the work presented here, re-training a model will require only a fraction of the effort compared to the alternative of modeling the permutation-performance behavior, a purely analytical solution, similar to one of the recent works presented by Li et. al [18].

For Conv-FP, we have $N$ loop at the outermost position and $C$ loop innermost, enforced by the analytical heuristic and these two loop positions are invariant across all permutations exposed to the predictive model. With that in mind, interesting observations can be made with regard to the permutations in the predominantly green region, towards the bottom end of the color map. All permutations in this region has $K_{16}$, the outer $K$ tile-controlling loop that results from vectorizing $K$, at the second innermost position. With $C$ at innermost position, $K_8C$ combination as the two innermost loops helps maximize both input and output tensor reuse. More precisely, $C$ loop carries the dependence for the output tensor, hence it has temporal reuse, and spatial reuse for the input tensor. Loop $K_{16}$ has spatial reuse for both output and weight tensors. Conversely, we notice the permutations associated with low-performing variants (red bands in the color map) carry either $P$ or $Q$ loops as the second innermost loop, simultaneously pushing $K_{16}$ towards the outer region of the loop order, subsequently hindering the output tensor’s chances at locality exploitation.
Furthermore, it is clear by comparing the two color maps (left and right) where the green and red regions overlap, that the model has learned to successfully identify these permutation clusters. This is further indication of the model’s ability to correctly map an arbitrary loop permutation with its expected performance.

5 Related Work

Compiler developers have traditionally applied classical machine learning methods like linear regression, Multivariate Adaptive Regression Splines (MARS), supervised classification, and genetic programming, amongst others [23] to compiler optimization problems. Advances in machine learning have allowed application of sophisticated ML techniques to compiler autotuning [6] providing effective solutions to otherwise intractable optimization problems such as phase ordering [14], auto-vectorization [13], schedule generation [26], optimal mapping [9], and others. Earlier application of ML primarily used static features, i.e., features specified at compile time [11]. Recent approaches use dynamic, i.e., architecture-independent features collected through dynamic profiling [7]. Hybrid strategies have also been proposed which use a combination of both [15]. We determined that for finding optimal loop permutation using our methodology, the use of static features suffices.

Related to our work, AutoTVM [8] aims to be competitive with hand-optimized code and defines an exhaustive schedule-space covering all hardware-aware optimizations in hand-tuned libraries and then focuses on efficiently finding an optimal schedule. Without focusing on improving the schedule space, the focus is on schedule optimization. The schedules search space includes schedules corresponding to multi-level tiling, loop ordering, shared memory caching, and annotations such as unrolling and vectorization. For each iteration, a set of top candidates has to be picked for a query. Top candidates are chosen using simulated annealing with a custom objective function as the energy function, instead of an exhaustive search. Recent work optimizes Halide [2] with tree search and use of random program snippets for tuning the cost model. Schedules corresponding to a combination of tiling, parallelization, and vectorization, are explored using tree search. Efficiency is achieved by putting some constraints on values of each configuration like tiling factor, loops to parallelize, and vectorization width, to prune the search space. Cummins et al. introduced DeepTune [9], an end-to-end approach for deep learning of optimization techniques. While their use of supervised learning to determine compiler heuristics is similar to ours, our work focuses exclusively on loop nest optimizations. FlexTensor [26] introduced a schedule exploration and optimization framework for tensor computation. It leverages a combined heuristic (simulated annealing) and RL (Q-Learning) method to find an optimized schedule involving multi-level loop tiling, loop reordering, loop parallelization and memory customization. One aspect in which our approach differs from the aforementioned end-to-end compiler auto-tuning approaches is by avoiding an online search to pick an optimization. Our ML model picks the most optimal loop permutation for a given loop nest as part of the inference step.

NeuroVectorizer [13] used deep reinforcement learning (RL) for end-to-end automatic vectorization. This work uses automatic feature selection by employing code2vec [3], a code embedding technique to automatically learn a function that maps loop codes to a feature vector. Our work uses hand-picked features that capture the pertinent characteristics of a loop nest and thus enable learning of better predictions through a simpler supervised regression model.

6 Conclusion

In this work we have demonstrated the viability of using machine-learning based models in conjunction with traditional compiler analysis. Specifically, we focused on the problem of choosing an optimal loop ordering for a class of problems with a relatively deep loop nest, and a complex data dependence pattern, which makes it hard for static compiler analysis. Our machine learning model accurately correlates the performance of a code variant with its loop order and is used to derive near optimal loop orderings in conjunction with other traditional loop optimizations such as unrolling and vectorization. The model achieves around 75% accuracy for the 2D convolution operators in predicting the performance of unseen loop orderings within a tolerance of around 5-10%. Further, the performance of the generated code, utilizing the model outperforms the state-of-the-art library oneDNN for matching data layouts, NHWC/RSCK with average Conv-FP speedup 1.67× and Conv-WU speedup 1.41×. Furthermore, we achieve performance within ~ 90% of oneDNN’s best implementation, which includes data layout transformations, for both Conv-FP (max 1.72× and 1.09×) and Conv-WU (max 0.95× and 0.89×).

7 Potential Broader Impact

The system presented in this work, ml4copt, aims to automate certain optimization decisions involved in the loop nest optimization process. Traditionally, this process is carried out by expert compiler developers and engineers. The rapidly increasing interest and success in automation in this area pose the possibility of removing human involvement from the loop nest optimization pipeline. However, on the positive side, these learned models can help improve the productivity of compiler developers by providing them with necessary guidance, significantly reducing the time spent on labor-intensive manual loop nest optimization process.

Being a deep neural network based model, the leaned solution lacks interpretability and therefore, human developers might find it non-intuitive to reason about the optimization
decisions made by the model. Furthermore, a potential adversarial party could use ml4opt to retrain the models with biased training sets in order to demonstrate fabricated results. For an example, one could craft a dataset that would lead to biased models whose predictions could undermine a rival vendor’s applications and/or hardware platforms. This can be avoided by carefully setting standards for fairness of the training sets and model evaluation criteria.

Acknowledgments
This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program under the RAPIDS Subcontract Award No. 4000159989, and by the National Science Foundation under Grant No. SHF-1564074.

Furthermore, authors extend their gratitude to Intel Academic Compute Environment (ACE) for the computational time and resources provided to successfully conduct the experiments presented in this paper.

References

