Efficiently Learning Locality Optimizations by Decomposing Transformation Domains

Tharindu Patabandi and Mary Hall (advisor)
University of Utah

ABSTRACT

Optimizing compilers for efficient machine learning are more important than ever due to the rising ubiquity of machine learning. Predictive models to guide compiler optimization are sometimes used to derive a sequence of loop transformations to optimize memory access performance via deploying learned models. However, training models for loop transformation often requires prohibitively expensive training data generation when predicting the combined effects of a transformation sequence. In this paper, we present a learning strategy called **Composed Singular Prediction** that significantly reduces the training data generation cost in the context of learned loop transformation models. The learned models are then deployed to predict data locality optimization schedules for Conv2d kernels to achieve **performance improvements up to 4.0× against Intel oneDNN** while **saving >100× in training data collection time**.

INTRODUCTION

- Learned models for compiler optimization are popular
- Datasets are not readily available
- Training data are generated on a case-by-case basis
- Requires some form of **Design Space Exploration (DSE)**
- Often requires sampling and/or pruning to handle the search complexity

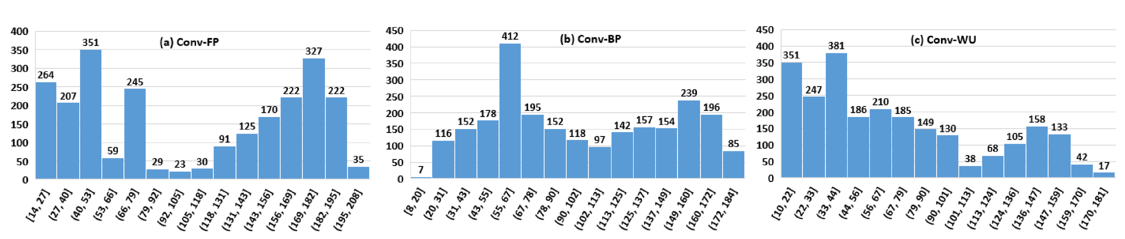


Fig 1. Performance distribution of loop permutation for Conv2d

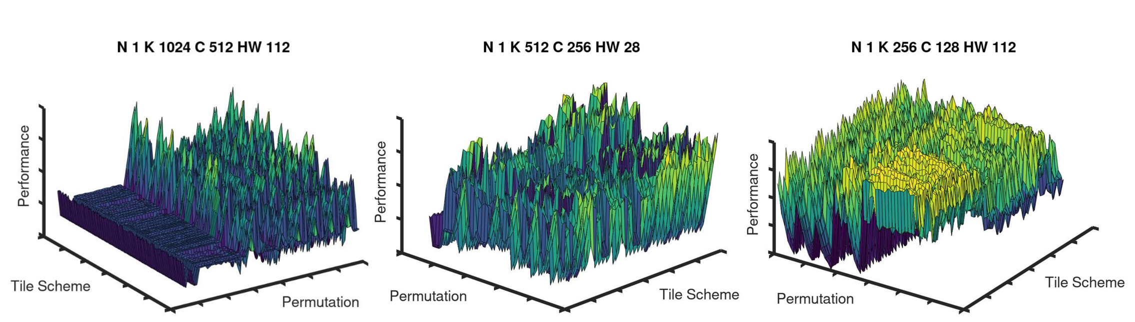


Fig 2. Performance distribution of loop tile and loop permutation for Conv2d

Conv2d in MLIR

Inputs are written in **affine** and **std** dialects. A custom **LLVM** pass consumes the MLIR input, applies loop transformations, and generate ***.llvm** for the selected architecture.

```
#input_map = affine_map<(d0, d1)[u] -> (u * d0 + d1)>

func @conv_forward(%A: memref<?x?x?x?xf32>,
                   %B: memref<?x?x?x?xf32>, %C: memref<?x?x?x?xf32>, %CH: index,
                   %K: index, %N: index, %P: index, %Q: index, %R: index, %S: index)
{ %u = constant 1 : index %v = constant 1 : index
affine.for %n = 0 to %N {
  affine.for %k = 0 to %K {
    affine.for %c = 0 to %CH {
      affine.for %p = 0 to %P {
        affine.for %q = 0 to %Q {
          affine.for %r = 0 to %R {
            affine.for %s = 0 to %S {
              %upr = affine.apply #input_map(%p, %r)[%u]
              %vqs = affine.apply #input_map(%q, %s)[%v]
              %ld.a = affine.load %A[%n, %upr, %vqs, %c] : memref<?x?x?x?xf32>
              %ld.b = affine.load %B[%r, %s, %c, %k] : memref<?x?x?x?xf32>
              %ld.c = affine.load %C[%n, %p, %q, %k] : memref<?x?x?x?xf32>
              %mul.ab = mulf %ld.a, %ld.b : f32
              %sum.ab.c = addf %ld.c, %mul.ab : f32
              affine.store %sum.ab.c, %C[%n, %p, %q, %k] : memref<?x?x?x?xf32>
           } } } } } } }
return }
```

METHODOLOGY

Search space for training data generation consists of different transformation schedule instances.

$$s = \{s_1, s_2, \dots, s_N\}, s_1 \in S_1, \dots, s_N \in S_N$$

Classical search space, **Multiplicative Domain Formulation (MDF)**

$$MDF: \{S_1 \times S_2 \times \cdots \times S_N\} \rightarrow O(|S_1| \times \cdots \times |S_N|)$$

With MDF, the learning task is to find a function f such that,

$$s^* = argmax_{s \in S} f(c, s)$$

Performance of solution schedule is, $\psi_{c,s^*} = \psi(\tau(c, s^*))$.

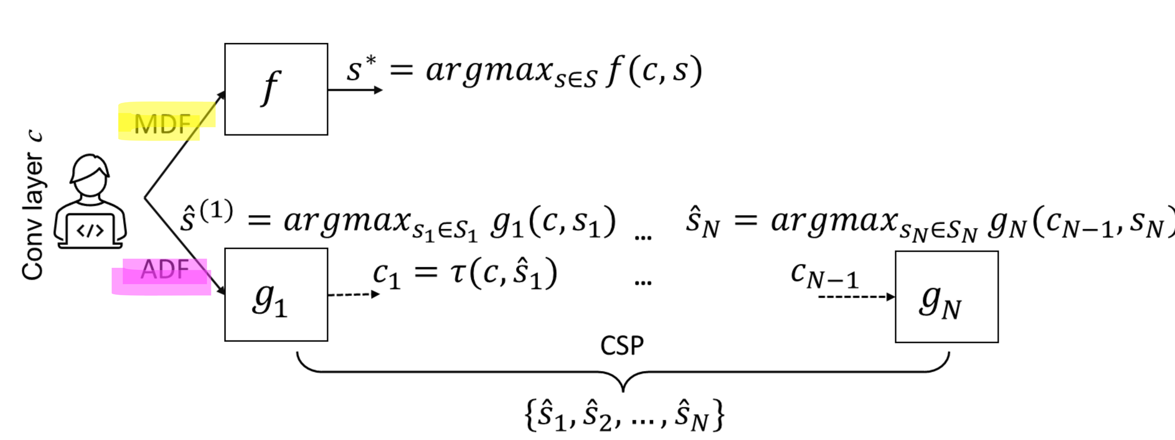
An alternative search space **Additive Domain Formulation (ADF)** can be defined as,

$$ADF: \{S_1 \cup S_2 \cup \cdots \cup S_N\} \rightarrow O(|S_1| + \cdots + |S_N|),$$
$$\because S_i \cap S_j = \{\phi\}\ \forall i \neq j$$

With ADF, the task is to learn a set of **Singular Functions** $\{g_k\}$ such that,

$$\hat{s} = \{\hat{s}_i: \hat{s}_i = argmax_{s \in S_i} g_i(c, s)\}$$

$$\underbrace{\psi_{c,\hat{s}} = \psi(\tau(\dots\tau(c, \hat{s}_1), \dots, \hat{s}_N))}_{\text{Composed Singular Prediction}}$$



CASE STUDY

- Data locality optimization for Conv2d kernel
- Loop tiling and Loop permutation with 2 Singular models
- MLIR/LLVM-based compiler
- Static loop unroll and vectorization
- Generates code for Intel Xeon AVX-512 systems

The permutation model computes,

$$p^* = argmax_{p \in S_2} g_2(c, s)$$

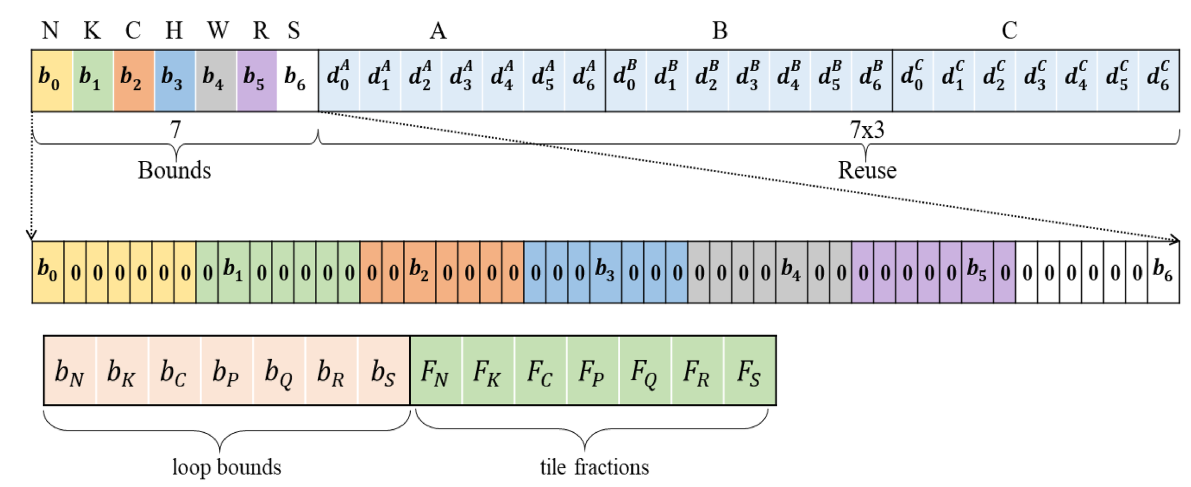
The tile model computes,

$$s^* = argmax_{s \in S_1} g_1(c, s)$$

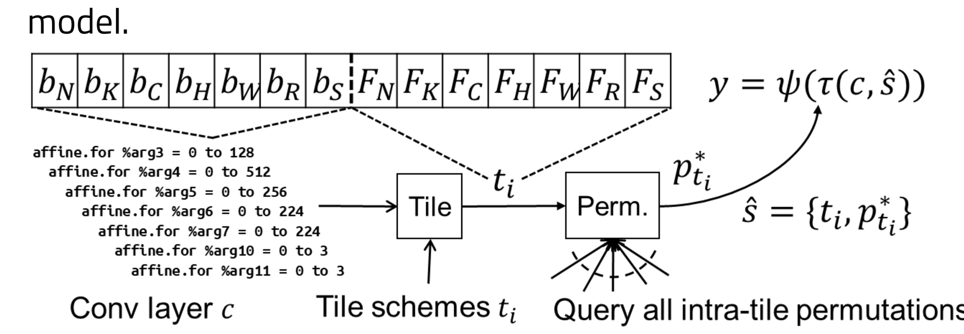
Performance of Composed Singular Prediction for the 2 transformations,

$$\psi(\tau(\tau(c, s^*), p^*))$$

Feature Representation

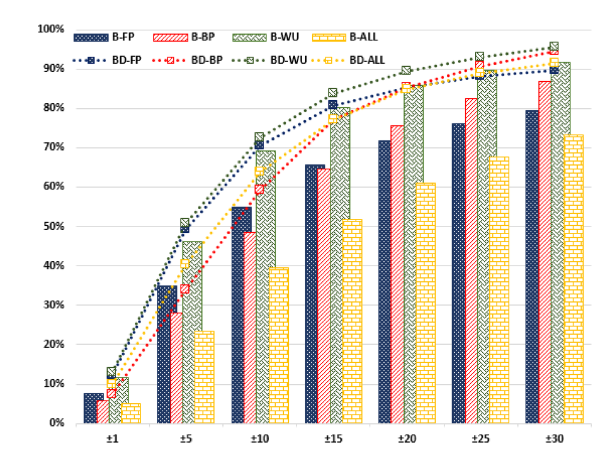


Training data generation for the tile model queries the loop permutation model.



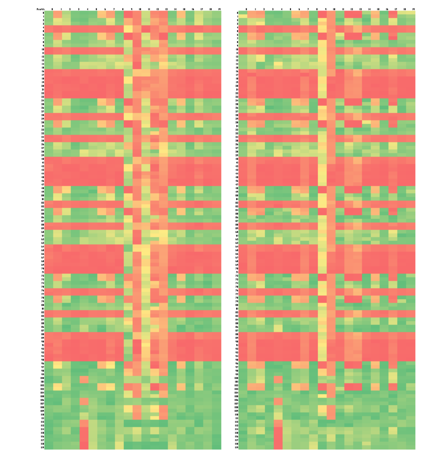
RESULTS

Loop Permutation Singular Model Accuracy



Accuracy of the Permutation model, measured in terms of varying error tolerance bounds.

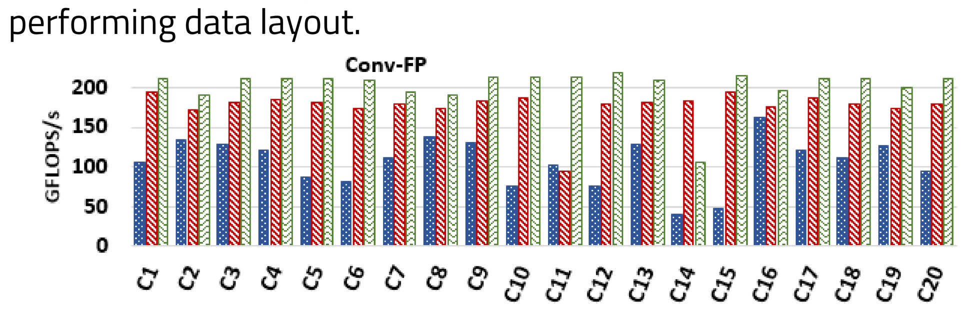
The performance value distributions of permutation model's predictions and its corresponding labels for all permutations of a select set of Conv2d layers.



(Left) Predictions (Right) Labels

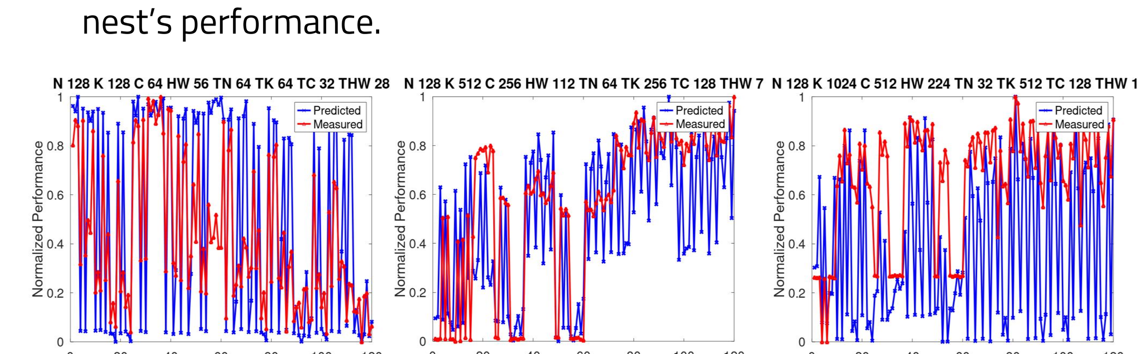
Performance of predicted permutation schedules

Performance of permutation schedules are compared against Intel oneDNN's matching data layout implementation and the library's best performing data layout.



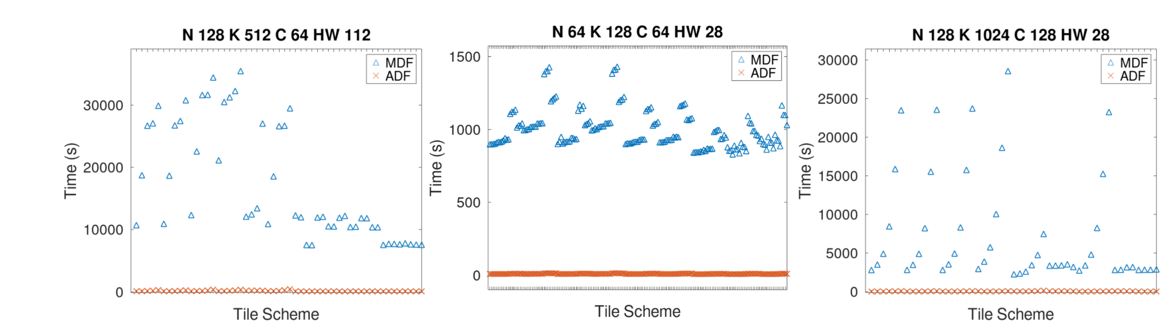
Intra-tile Performance Approximation

Performance of a tiled Conv2d is approximated with its intra-tile sub-nest's performance.



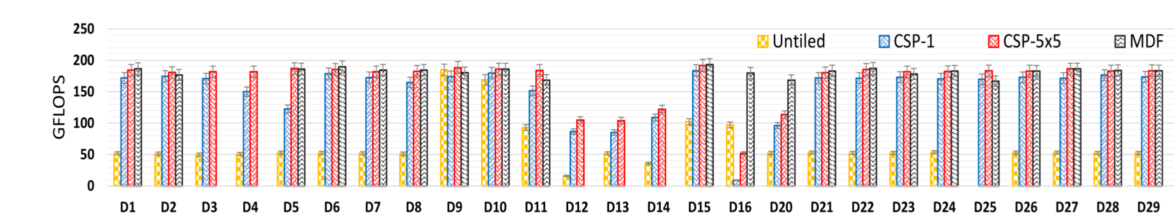
Training Data Collection Time with ADF

Time spent evaluating the performance of all intra-tile permutations of a given tile scheme, compared to its corresponding MDF data collection.

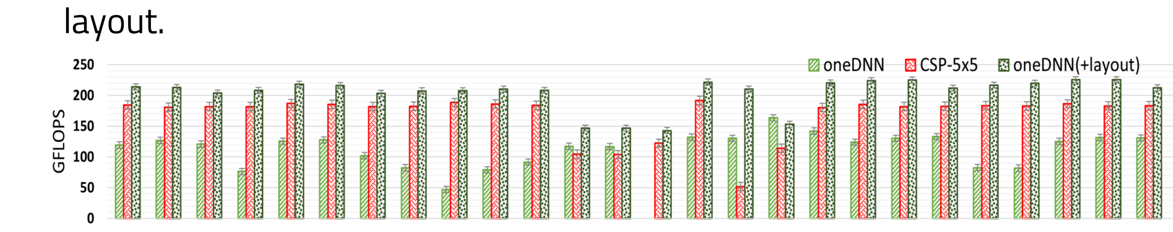


Performance of predicted tile schedules

Performance of CSP-1 and CSP-5x5 schedules are compared against each large Conv2d layer's best untiled performance and MDF variant.



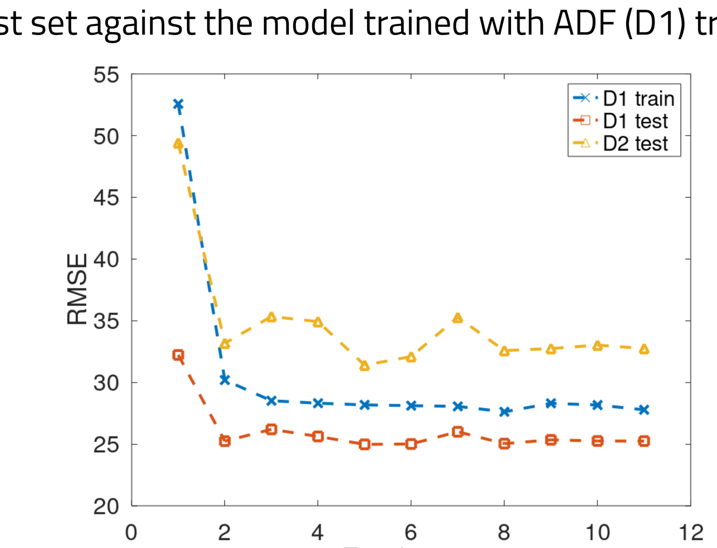
Performance of CSP-5x5 schedules are compared against Intel oneDNN's matching data layout implementation and the library's best performing data layout.



Performance speed-ups of the tile model's predictions against untiled Conv2d and Intel oneDNN library implementation.

	Untiled		oneDNN	
	AVG	MAX	AVG	MAX
CSP-1	2.4x	5.5x	1.4x	3.7x
CSP-5x5	2.7x	6.6x	1.5x	4.0x

Further evidence for domain correlation is observed by evaluating a small MDF (D2) test set against the model trained with ADF (D1) training data.



ACKNOWLEDGEMENTS

This work began in Summer 2020 at Intel Labs Machine Programming Research (MPR) Group.
Anand Venkat, Justin Gottschlich (Intel Labs)
Abhishek Kulkarni, Pushkar Ratnalikar (Intel)
Vivek Srikumar (University of Utah)
Intel Academic Compute Environment (ACE)

Funding
DOE SciDAC, NSF