Performance Embeddings: A Similarity-Based Transfer Tuning Approach to Performance Optimization

Lukas Trümper
lukashans.truemper@inf.ethz.ch
ETH Zurich
Switzerland

Tal Ben-Nun∗
talbn@llnl.gov
Lawrence Livermore National Laboratory
USA

Philipp Schaad
philipp.schaad@inf.ethz.ch
ETH Zurich
Switzerland

Alexandru Calotoiu
alexandru.calotoiu@inf.ethz.ch
ETH Zurich
Switzerland

Torsten Hoefler
htor@inf.ethz.ch
ETH Zurich
Switzerland

ABSTRACT
Performance optimization is an increasingly challenging but often repetitive task. While each platform has its quirks, the underlying code transformations rely on data movement and computational characteristics that recur across applications. This paper proposes to leverage those similarities by constructing an embedding space for subprograms. The continuous space captures both static and dynamic properties of loop nests via symbolic code analysis and performance profiling, respectively. Performance embeddings enable direct knowledge transfer of performance tuning between applications, which can result from autotuning or tailored improvements. We demonstrate this transfer tuning approach on case studies in deep neural networks, dense and sparse linear algebra compositions, and numerical weather prediction stencils. Transfer tuning reduces the search complexity by up to four orders of magnitude and outperforms the MKL library in sparse-dense matrix multiplication. The results exhibit clear correspondences between program characteristics and optimizations, outperforming prior specialized state-of-the-art approaches and generalizing beyond their capabilities.

CCS CONCEPTS
• Software and its engineering ➔ Compilers; • Computing methodologies ➔ Machine learning.

KEYWORDS
compilers, embeddings, transfer tuning, peephole optimization, performance optimization, autotuning

ACM Reference Format:

1 INTRODUCTION
Automatic performance optimization of programs for modern computing architectures is challenging. Even for smaller programs, the possibilities to schedule the operations and the data movement become infeasible to explore exhaustively. To efficiently navigate the optimization space, a performance model could be constructed as a surrogate to approximate the search; the searched parameters can be limited to a small number for brute-force tuning; or, more often than not, the program is optimized manually by a performance engineer.

Several performance models have been developed for specific program classes, notably Polyhedral subprograms [12]. The polyhedral model has helped develop several automated tuning methods based on integer-linear programming [4] and machine learning [1, 5, 18] as well. Such methods primarily target optimizations on the loop level such as interchanging their order and tiling the iteration space. However, these techniques are limited in representing real-world applications due to the need for expressing programs with affine array accesses and simple loop bounds.
Methods for optimizing data-dependent applications, such as sparse linear algebra routines, must rely on specialized, input-specific models [27]. Because such models are hard to integrate into a general tuning framework, performance engineers often fall back to general profiling-based performance models, such as the roofline model [61], for custom applications. Since profiling-based models lack a connection to the algorithmic structure, their interpretation requires significant experience [55], which makes the search for optimizations hard to automate. Optimization efforts for real-world applications are thus often resource-intensive manual processes where the outcome strongly depends on the skill set of the individual performance engineer [8, 16, 54].

In this paper, we present a similarity-based approach to the automatic performance optimization of general loop nests, summarized in Figure 1. We develop a method for encoding both static and dynamic performance characteristics of loop nests and capturing them as performance embeddings — a latent, continuous space in which a multidimensional point represents a subprogram. Based on these embeddings, which are trained separately, optimizations derived from a variety of methods (such as brute force, manual tuning, or state-of-the-art auto-schedulers) are stored in an optimization database. This enables knowledge transfer of optimization between different programs with similar static or runtime characteristics, which we call transfer tuning.

During transfer tuning, loop nests are then optimized by fuzzy matching the optimizations of the k-nearest neighbors from the database according to their performance embeddings. We demonstrate the effectiveness of our approach on a series of polyhedral and non-polyhedral real-world applications, significantly reducing the search complexity for performance optimizations and outperforming state-of-the-art auto-schedulers by reaching up to 92% better runtime improvements.

In summary, this paper makes the following contributions:

- Methodology for encoding performance characteristics of general loop nests in performance embeddings;
- Development of a general matching algorithm for loop nest optimizations;
- Reduction of the optimization search space size by orders of magnitude through transfer tuning;
- Demonstration of effectiveness compared with state-of-the-art auto-optimizers and extension to tailored optimizations.

2 SIMILARITY IN PERFORMANCE OPTIMIZATION

Programs with different structural properties may still share similar performance characteristics, which allow them to be optimized in similar manners.

The following example shows a standard matrix multiplication and a min-plus matrix multiplication commonly used for shortest-path problems:

```c
#pragma omp parallel for
for (int i=0; i < 1024; i++)
for (int j=0; j < 1024; j++)
    x[i] += val[k] * y[col[k]]

#pragma omp parallel for
for (int i=0; i < 1024; i++)
for (int j=0; j < 20000; j++)
    if (numbers[i] == 1) {
    is_prime[i] = false;
    } else {
    is_prime[i] = true;
    for (int j=2; j < sqrt(numbers[i]); j++) {
        if (numbers[i] % j == 0) {
            is_prime[i] = false;
            break;
        }
    }
}
```

Compared to the regular, dense matrix multiplication from before, this loop nest is no longer data-oblivious since the innermost loop bounds are data-dependent. The sparsity pattern of the input thus determines the workload’s characteristics (e.g., load balancing over multiple threads). Regardless of those characteristics, both programs exhibit a strided memory access to the dense matrix B, which can be resolved by interchanging the two innermost loops to improve performance. Existing auto-schedulers [1, 5] can apply this optimization to the original matrix multiplication, but can only transfer these optimizations to the sparse multiplication if their performance models indicate similar performance characteristics. Such static models must, however, make simplifying assumptions on the code, either assuming a fixed sparsity pattern and over-approximating the loop bounds [10], or using an inspector-executor model [53] to produce code conditionally. Both assumptions hinder possible further optimizations with regard to load imbalance and dynamic characteristics.

A case where the structural differences are even more pronounced is shown below, where the first program computes a sparse matrix-vector product, and the second program performs a prime number check on an array of 20,000 numbers:

```c
#pragma omp parallel for
for (int i=0; i < 1000000; i++)
    c[i][j] += a[i][k] * b[k][j]

#pragma omp parallel for
for (int i=0; i < 20000; i++)
    if (numbers[i] == 1) {
        is_prime[i] = false;
    } else {
        is_prime[i] = true;
        for (int j=2; j < sqrt(numbers[i]); j++) {
            if (numbers[i] % j == 0) {
                is_prime[i] = false;
                break;
            }
        }
    }
```

Despite their structural differences, both programs are inherently prone to an imbalanced distribution of work among different threads when parallelizing the outermost loop. In both cases, a dynamic assignment of work to threads yields significantly better performance for specific input distributions. While a purpose-built, data-specific model [27] can address this problem for the sparse matrix-vector product, the same model cannot directly be applied to the
structurally-different prime number filter. Hence, in order to identify similarities and transfer optimizations between data-dependent applications, the integration of a larger number of specialized models would be necessary.

In contrast, performance engineers are able to identify similarities between both data-oblivious and data-dependent applications treating data-dependent aspects as gaps, which are inferred through profiling. Performance embeddings adopt this observation by encoding both static and dynamic performance characteristics of parallel loop nests, enabling the transfer of optimizations across more general problems.

3 EMBEDDING PARALLEL LOOP NESTS

The basis of the similarity search is a representation of parallel loop nests which captures a rich set of performance-relevant properties. This representation should encode static properties such as the structure of loops, and the data accesses, but also reflect dynamic properties such as the bandwidth utilization, the thread imbalance, or the amount of mispredicted branches. In contrast to approaches solely focusing on runtime prediction for data-oblivious applications [1, 5, 51], the purpose of this representation is to provide a detailed description of performance for general parallel loop nests; the runtime itself does not expose information about the potential for optimization.

We compute the representation of parallel loop nests using neural networks based on both static and dynamic features, depicted in Figure 2. Dynamic features (performance counters) measured on representative inputs allow the model to treat input-specific aspects of a parallel loop nest as gaps in the static analysis. These features inform the model about the behavior of the loop nest via hardware metrics. For example, the load imbalance between threads is a direct result of a matrix’s sparsity pattern in a sparse matrix multiplication.

3.1 Parallel Loop Nests

Before introducing the representation, the term parallel loop nest shall be defined in detail. A parallel loop defines a parallel iteration space and a (possibly empty) body of computations executed for each iteration. A parallel loop nest is an ordered tree where each node is a parallel loop nested inside the iteration space of the parent. A program is considered a set of parallel loop nests, which are optimized independently. This assumes that optimizations on the full program have been determined beforehand, e.g., the identification of parallelism and the fusion of parallel loop nests. A fusion strategy based on similarity is briefly discussed in Section 8.

The computations and loop extents are not assumed to be known at compile-time. In particular, the body may comprise sequential loops and recursions whose function depends on input data. Compared with other models [1, 5], this definition relaxes the requirements of compile-time known loop extents, operations, and memory access patterns.

3.2 Encoding

The encoding maps the parallel loop nest given in an intermediate representation (IR) to a set of features, which can be processed by a neural network. The encoding of parallel loop nests consists of two parts: a graph encoding of the static IR and an encoding of the dynamic profiling information in a single vector. A detailed list of the used static and dynamic features is presented in Appendix A.

Static Encoding. The basis of the static encoding is a parallel loop nest represented as a stateful dataflow multigraph (SDFG) [7]. SDFGs combine state machines with dataflow graphs to represent complete programs, which makes them amenable for static analysis and simplifies the mapping to a graph encoding. However, the approach could equally be implemented with other IRs, e.g., LLVM IR [41].

At the outermost scope, the SDFG of a parallel loop nest is a dataflow graph comprising at least a single parallel loop, called map. As shown in Figure 3, the body of the map may comprise nested maps, tasklets (operations), or nested SDFGs. The components of an SDFG are mapped to a graph of nodes with features and edges as follows:

- Access node: Access nodes represent data in the data-flow graph and are mapped to corresponding nodes in the encoding. These nodes are represented by features such as shape, total size, data type, and data layout.
As illustrated in Figure 2, the two encodings are first processed in separate branches of the neural network. A linear embedding layer maps the dynamic encoding to a dynamic embedding. A graph neural network (GNN) based on the graph transformer operator [49] maps the static encoding to node embeddings, which are summarized into a graph embedding by an attentional pooling layer [43]. Finally, the graph embedding is concatenated with the dynamic embedding and mapped by another MLP to an embedding of the entire parallel loop nest. The size of the embeddings is fixed to 128 for node and graph embeddings. In total, the model comprises 44 layers and 862,000 trainable parameters. The implementation of the model is written in PyTorch Geometric.

**Targets and Training.** To train the model, we add another linear layer to the model, which predicts a target vector based on the embedding of the parallel loop nest. These targets comprise 20 standard performance metrics of the parallel loop nest summarized in Figure 4. This includes the runtime, the instructions per cycle, different bandwidths, cache miss ratios, and several rates of specific operations per total instructions. We choose the mean absolute error as the loss function and train the model for 20 epochs using Adam at a learning rate of 1e−3. We do not specifically tune the hyperparameters of the model beyond manually setting an initial learning rate, and use an early stopping approach for the weights.

**Dataset.** We synthetically generate the training and validation set from standard kernels such as maps, reductions, and stencils. In particular, we include non-data-oblivious kernels such as boolean masks. The test set is extracted from real-world applications implemented in NPBench [63] by automatically cutting out each parallel loop nest. The sizes of the training, validation, and test sets cover approximately 6,590, 2,000, and 1,000 parallel loop nests, respectively. In contrast to other models designed to predict the speedup of different schedules, we consider a single canonical schedule, which significantly reduces the input variation. The canonical schedule executes the outermost loop of the loop nest in parallel.

**Target Architecture.** The target architecture is an Intel Xeon Gold 6140 CPU with a base clock rate of 2.3 GHz and 768 GB of main memory. The entire dataset is labeled automatically with LIKWID [55], which defines groups of performance metrics that can be measured simultaneously. Each group of metrics is measured in two phases: In a warmup phase, the program is executed \( n_w \) times, where \( n_w \) is chosen such that the logical number of bytes moved corresponds to twice the size of the L2 cache but clipped to a maximum of 1,000 repetitions. In the measurement phase, the program is executed ten times and the median is taken over those measurements to convert
the measurements into a single label. In general, most metrics report the measured mean over all threads. However, global throughput metrics such as bandwidths or the instruction per cycle are summed over the threads; the runtime is considered as the maximum over all threads.

3.4 Validation
Before evaluating the quality of embeddings on application-specific tasks, we validate the model on the prediction of the performance metrics. Figure 4 lists the Pearson correlation coefficient between the targets and the model’s predictions on the test set for the different performance metrics. In this figure, the performance metrics are ranked by their difficulty of prediction by our model in descending order. The minimum correlation of 0.60 is found for Instructions Per Cycle and the maximum correlation of 0.98 for the metric of Dropped Cache-Lines Bandwidth. For 17 out of 20 targets, the correlation is at least 0.80, indicating a strong correlation between the model prediction and the target labels. These results also correlate with the difficulty of prediction in general, as, e.g., the Instructions Per Cycle metric depends on multiple hardware and system factors.

4 PERFORMANCE SIMILARITY
A similarity search for performance optimization requires that similar embeddings imply similar performance optimization potentials. For instance, if a parallel loop nest has a low memory bandwidth utilization, this loop nest should be mapped to an embedding that is similar to the embeddings of other parallel loop nests with low memory bandwidth utilization.

We evaluate this hypothesis based on the local variation of parallel loop nests under different performance metrics. Specifically, for each parallel loop nest in the test set, we query the 3-nearest-neighbors based on the embedding distance and compute the relative standard deviation among these four loop nests for a specific performance metric. We define the mean of the local variations in the test set as the performance similarity of the model.

Below, we discuss the similarity metrics we use for our evaluation, the state-of-the-art baselines we compare with, and analyze similarity on the NPBench dataset.

Assessing similarity. Since the cost for data movement is the dominant factor in performance optimization [56, 57], we focus on memory-specific performance metrics for evaluation. The memory usage efficiency (MUE) [29] combines the following two performance metrics to assess the optimization potential of a program:

- **Main / L3 / L2 Memory Bandwidth**: The attained memory bandwidth on different levels of the memory hierarchy is a standard metric to identify optimization potentials in typical bound-and-bottleneck analyses (cf., Roofline model [35, 61]).
- **Data Locality**: Fuhrer et al. [29] point out that an analysis based on solely the attained memory bandwidth ignores the intrinsic limitations of the algorithm. For instance, a loop nest with a strided memory access pattern and a loop nest with a random memory access pattern may both yield low memory bandwidths. However, the former may still be optimized through a loop interchange, while the latter already achieves its maximal bandwidth utilization. The data locality accounts for these algorithmic limitations and is defined as the ratio of the I/O lower bound $Q$ of the algorithm and the measured transferred bytes from main memory $D$. In short, $\frac{Q}{D}$. $Q$ is estimated automatically by SOAP-Analysis [40], which is based on the concept of the Red-Blue Pebble Game [37].

Baselines. To assess the model’s performance, we compare the similarity of our embeddings with three other models that map parallel loop nests to embeddings, and perform ablation studies on the input features.

The reuse distance analysis [11, 19, 48] is a traditional approach to loop nest analysis, which simulates the execution of the loop for a specified number of iterations on a simplified cache model. Using this simulation-based analysis, we map each loop nest to a four-dimensional vector of the cache miss ratio, the bytes read from and written to the memory, and the arithmetic intensity. The movement of bytes gives a strong indication of the efficiency of the memory access patterns, and the arithmetic intensity is typically used to estimate the performance of a program on a target architecture. Since the simulation of loop nests is expensive, we only simulate the first 500 iterations of the loop nest.

IR2Vec [60] provides embeddings of programs based on LLVM IR. The embeddings are trained in an unsupervised manner and can be used for various machine learning tasks related to program properties and source code.

Baghdadi et al. [5] introduce a state-of-the-art performance model for optimizing polyhedral programs. The model estimates the speedup of a schedule and a loop nest based on static features and a recurrent neural network. Since the model is designed to predict the speedup of a certain schedule, we remove the linear prediction layer and obtain the embedding of the parallel loop nest from the input of this last layer.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Bandwidth</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Main</td>
<td>L3</td>
</tr>
<tr>
<td>Reuse Distance [11, 19]</td>
<td>0.78</td>
<td>1.02</td>
</tr>
<tr>
<td>IR2Vec [60]</td>
<td>0.47</td>
<td>0.66</td>
</tr>
<tr>
<td>Baghdadi et al. [5]</td>
<td>0.32</td>
<td>0.41</td>
</tr>
<tr>
<td><strong>Our Model</strong></td>
<td><strong>0.25</strong></td>
<td><strong>0.30</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>Bandwidth</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Features</td>
<td>0.26</td>
<td>0.33</td>
</tr>
<tr>
<td>Static Features</td>
<td>0.28</td>
<td>0.42</td>
</tr>
</tbody>
</table>

Table 1: The mean coefficient-of-variation of different feature extractors on the test set. A lower value means higher similarity among the three closest neighbors.

Results. Table 1 summarizes the performance similarity of the baseline feature extractors and our model. Our model has a strictly lower local variation for all performance metrics and thus yields a higher performance similarity. Hence, the performance optimization based on the local neighbors in our embedding space is more likely to resolve the actual performance bottlenecks of a parallel loop nest. Furthermore, we run ablation studies using only one set of the static/dynamic features. The studies show that the selected dynamic features are sufficient for reasoning over bandwidth.
However, static features (such as array accesses) are crucial to understand memory access patterns for data locality and I/O complexity.

Similarly to text analogies for word embeddings, we additionally verify our representation through the use of several distance tests. For example, one of our tests implements three operations: linear copy of two arrays (denoted as a), indirect copy with a random permutation on the indices (b), and indirect copy with the identity index permutation (c). In all of our learned embeddings, \( d(a, c) < d(a, b) \) for the cosine distance \( d \).

To further understand the similarity induced by our model, Figure 5 visualizes the embeddings of the test set in a t-SNE plot [58]. A t-SNE plot reduces high-dimensional data onto a 2D plane based on neighborhood minimization. In the figure, each sample is a point colored by its data locality; a plot that is separable by color, as our model’s embedding space is (Figure 5a), indicates a strong influence of the performance metric in the representation of the sample. For comparison, Figure 5b shows that the data locality is not an important factor for the representation of the sample, depicted by scattered clusters.

Evaluating importance of static features. Since the model has a rich set of dynamic features available, the question arises whether the static encoding is used by the model. To analyze this question, we analyze the structure of the node embeddings for the input.

5 TRANSFER TUNING

Peephole optimization is a compiler technique that replaces a local window of instructions with an equivalent set. Such local windows are usually found using a pattern-matching algorithm. However, since the replacement rules of peephole optimizations are designed for bit-exactness, the applicability of the optimization is limited to small windows of a few instructions. Our transfer tuning algorithm extends the idea of peephole optimizations to larger loop nests by fuzzy matching program transformations from one loop nest to another via similar node embeddings.

5.1 A Matching Problem for Program Transformations

Transferring a transformation from a source loop nest to a target loop nest requires identifying the corresponding instructions, to which the transformation shall be applied in the target. As an example, consider the pair of loop nests in Figure 7 and a transformation that marks a loop for parallel execution. To transfer it to the target loop nest, the corresponding loop must be identified. Since parallel loop nests are represented by graphs in our model, instructions correspond to nodes and edges of the graph. Furthermore, since the node embeddings generated by the model have a one-to-one correspondence with the nodes in the IR, transformations shall be transferred from the source to the target parallel loop nest by...
We now evaluate transfer tuning in two case studies: In the first case, a program transformation can thereby range from a simple change to an identical weather prediction, and linear algebra. In the second case, other examples are the optimizations from different domains such as image processing, numerical weather prediction, and linear algebra. In the second case study, dynamic scheduling decisions are transfer tuned between applications with a large scheduling space.

We evaluate transfer tuning in two case studies: In the first case, a program transformation can thereby range from a simple change to an identical weather prediction, and linear algebra. In the second case, other examples are the optimizations from different domains such as image processing, numerical weather prediction, and linear algebra. In the second case study, dynamic scheduling decisions are transfer tuned between applications with a large scheduling space.

56

6 EVALUATION

We now evaluate transfer tuning in two case studies: In the first case study, the optimizations found by a state-of-the-art auto-scheduler for polyhedral applications [5] are transfer tuned between applications from different domains such as image processing, numerical weather prediction, and linear algebra. In the second case study, dynamic scheduling decisions are transfer tuned between applications comprising approximately one hundred parallel loop nests.

6.1 Case Study: Auto-Scheduler

Baghdadi et. al. [5] train a speedup prediction model and use this model to guide the search of the Tiramisu auto-scheduler in a large scheduling space consisting of typical loop transformations such as loop interchange, tiling, parallelization, and vectorization. We show that transfer tuning the discovered optimizations between applications based on the performance embeddings reduces performance optimization to a local search. The evaluation set consists of 12 applications comprising approximately one hundred parallel loop nests.

Experimental Setup. To find a strong reference optimization for each parallel loop nest, we run the Tiramisu auto-scheduler’s Monte-Carlo Tree Search (MCTS) for a larger number of epochs. Additionally, we test the 100 best hypotheses found by the search on the target architecture to determine the overall best-performing configuration. Hence, the optimization database comprises approximately one hundred schedules corresponding to the total number of parallel loop nests. The transfer tuned optimization for a parallel loop nest is found by a k-nearest-neighbor search in the embedding space of all parallel loop nests except for the parallel loop nest to be tuned (leave-one-out). To apply the Tiramisu auto-scheduler to our graph IR, we implement a converter from SDFGs to the representation of programs used by this model.

Results. Table 2 lists the results of the Tiramisu auto-scheduler’s optimization of each application as well as the results obtained by transfer tuning for k = 5 and k = 10 neighbors. For the majority of applications, the transfer-tuned runtime is within 5% of the reference at a fraction of the search complexity, see MCTS Space column for the number of configurations tested by the auto-scheduler. Since the reference optimizations are found once and then stored in the database, transfer tuning enables exhaustive offline optimization of applications with a large scheduling space.

Daubechies Wavelet. In the embedding space, the neighbors of a parallel loop nest act as a collection of explored search paths based on slightly varied input conditions. The Daubechies wavelet benchmark is an example where this neighborhood yields a considerable speedup. The application consists of a single parallel loop nest, where the outermost loop iterates over the 3 channels of an image. Parallelizing over this loop induces a major performance bottleneck on a CPU with 36 cores since most cores are idling.

Upon inspecting the transferred transfer tuning results, we see that it optimized according to the Haar wavelet: MCTS fails to find an optimization maximizing the parallelism for the Daubechies wavelet, but succeeds in finding an optimization for the almost identical Haar wavelet. This also showcases an important feature of performance embeddings — as opposed to end-to-end neural networks, transfer tuning provides explainability for its optimization decisions.

Other examples are the Harris filter and the histogram filter, in which transfer tuning finds additional potential for applying the optimization found within the same benchmark.
<table>
<thead>
<tr>
<th>Application</th>
<th>Baghdadi et al. [5]</th>
<th>Transfer Tuning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MCTS Space</td>
<td>Runtime [ms]</td>
</tr>
<tr>
<td>Deep Learning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mlp</td>
<td>111,508</td>
<td>1.47</td>
</tr>
<tr>
<td>softmax</td>
<td>183,427</td>
<td>110.40</td>
</tr>
<tr>
<td>Image Processing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blur filter</td>
<td>1,342</td>
<td>1.03</td>
</tr>
<tr>
<td>daubechies wavelet</td>
<td>9,101</td>
<td>8.73</td>
</tr>
<tr>
<td>haar wavelet</td>
<td>8,639</td>
<td>0.22</td>
</tr>
<tr>
<td>harris filter</td>
<td>1,651</td>
<td>9.06</td>
</tr>
<tr>
<td>histogram filter</td>
<td>147,438</td>
<td>32.51</td>
</tr>
<tr>
<td>unsharpening filter</td>
<td>25,080</td>
<td>29.66</td>
</tr>
<tr>
<td>Weather Stencils</td>
<td></td>
<td></td>
</tr>
<tr>
<td>heat 3D</td>
<td>69,080</td>
<td>13428.98</td>
</tr>
<tr>
<td>horizontal diffusion</td>
<td>34,534</td>
<td>7.00</td>
</tr>
<tr>
<td>Linear Algebra</td>
<td></td>
<td></td>
</tr>
<tr>
<td>matmul</td>
<td>65,986</td>
<td>14.17</td>
</tr>
<tr>
<td>Graphs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>min-plus mm</td>
<td>65,999</td>
<td>24.76</td>
</tr>
</tbody>
</table>

Table 2: The runtime difference of transfer tuning for five and ten neighbors relative to the runtime of the Tiramisu auto-scheduler [5] for polyhedral applications. The auto-scheduler uses Monte-Carlo Tree Search (MCTS) to explore a large schedule space, whereas transfer tuning is a local search based on a few nearest neighbors.

Multi-Layer Perceptron (MLP). Although matmul and min-plus matrix multiplication are potential candidates for optimizing the layers in mlp, we see that transfer tuning performs worse for this particular benchmark. The matrix multiplications of matmul and mlp differ significantly in the dimensions of the matrices: while matmul multiplies a $1024 \times 2048$ and a $2048 \times 1024$ matrix, mlp multiplies weight matrices, which have a small leading dimension of 64 corresponding to the batch size. Hence, the matrix multiplications define different trade-offs of data locality and parallelization. This shows that the optimization database’s density (i.e., the availability of similar neighbors) is an important hyperparameter of transfer tuning.

6.2 Case Study: Tailored Optimization

In the second case study, we demonstrate the extensibility of transfer tuning to custom optimizations by dynamically scheduling SpMMs for matrices from suitesparse [23]. A typical performance bottleneck of SpMM is an imbalanced distribution of work among the threads, resulting from the distribution of the non-zero elements. The standard optimization is then to change the scheduling from a static assignment of work to threads to a dynamic assignment, which incurs some overhead for the execution.

Experimental Setup. To define an optimization database for the scheduling decision, we determine the optimal schedule for 42 sparse matrices from suitesparse [23] by benchmarking OpenMP’s default static schedule and a dynamic schedule of chunk size 8. The matrices are multiplied by a dense matrix of 512 columns filled with random values. We evaluate whether transfer tuning can decide the optimal schedule by splitting this set of matrices into a set that is stored in the optimization database and a test set. The scheduling of the test set matrices is then determined by a 1-nearest neighbor query to the database. The resulting runtime of the matrices is compared with the Intel MKL 2021.3 implementation of SpMM.

Results. The t-SNE plot of the SpMM embeddings of all matrices is depicted in Figure 9, where the embeddings of the different matrices are colored by their optimal scheduling type, i.e., static (purple ●) and dynamic (orange ●).

Figure 9: t-SNE plot of the SpMM embeddings for 42 suitesparse matrices. Embeddings are colored by the optimal scheduling type, i.e., static (purple ●) and dynamic (orange ●).
speedup of the optimal scheduling for a different subset of 8 out of 10 benchmarks.

**BERT.** The BERT transformer [25] is a standard neural network architecture in natural language processing. The sparsification of the dense layers is a common technique to enable efficient inference by sacrificing a reasonable amount of accuracy [34]. In order to show the cross-domain transfer of this knowledge, we repeat the above experiment for the sparse weights of a sparsified model [39], yielding a similarly separable embedding space for transfer tuning. The tSNE plot of the sparse weights is depicted in Figure 10.

In conclusion, transfer tuning yields comparable performance speedups on all tested cases, at times outperforming existing tools and libraries by inferring cross-application optimizations. It can adapt to additional insights gained by automated tools and tailored optimizations and can be inspected to explain its reasoning behind certain optimizations via the chosen neighbor.

### 7 RELATED WORK

Automatic performance optimization and performance modeling for optimization has been studied by a variety of works. The following section summarizes prior related research.

**Performance Modeling and Extrapolation.** Several works focused on the automatic prediction of program and subprogram performance. One of the earlier instances of using machine learning for performance modeling was performed by Ipek et al. [36], who use an MLP to predict application performance. Carrington et al. [17] and Siegmund et al. [50] also provide performance prediction for tuning via heuristic means on an application-level, and Calotoiu et al. [15] model and extrapolate runtime dependency on parameters of general codes via time measurement of multiple small experiments. Most such works do not focus on the optimization transformations and their choice, but rather on accurate execution time prediction.

Application-specific performance models [33, 42, 62] introduce domain knowledge into the prediction and often use the generated communication or performance model to inform an optimization search without executing the program, which might be expensive due to running on distributed environments.

**Polyhedral Compilers.** The Pluto [13], PENCIL [3], and LLVM Polly [32] compilers express performance optimizations as the solution of an integer linear program (ILP) with respect to a hand-crafted cost model of the target architecture. For reasons of tractability of the ILP, the cost model makes strong simplifying assumptions, often yielding sub-optimal results on complex architectures [4].

**Deep Code Representations.** Neural code representations that map static code to embeddings. The embeddings are designed to solve typical compiler tasks and classify applications according to their semantics. In contrast, performance embeddings encode static and dynamic properties, aiming to capture performance aspects regardless of the underlying algorithm.

**Optimizing Compilers.** Optimizing compilers are subject to extensive research. Tiramisu [6], Halide [47] and TVM [18] introduce deep learning performance models [1, 5, 18] based on static features, which guide the search in the scheduling space. Singh et al. [51] extends these performance models to graph neural networks, improving the prediction’s accuracy. Steiner et al. [52] re-formulate the search problem as a Markov Decision Problem, which can be solved using reinforcement learning. Other works utilize input-specific and profiling features to optimize programs based on classification problems: For instance, Elafrou et al. [27] train a neural network to choose between classes of optimizations for sparse linear algebra routines. Dutta et al. [26] combine a pattern classifier and performance counters for selecting OpenMP configurations. Our approach separates the performance model from the optimization by introducing an offline optimization database. This allows the local search in the application space, which significantly reduces the complexity of the search and allows for the extension of the optimization space without re-training the model. In particular, our

<table>
<thead>
<tr>
<th>Sparse Matrix</th>
<th>Static</th>
<th>Dynamic</th>
<th>Transfer</th>
<th>MKL</th>
</tr>
</thead>
<tbody>
<tr>
<td>as-Skitter</td>
<td>2574.19</td>
<td>719.31</td>
<td>719.31</td>
<td>1264.84</td>
</tr>
<tr>
<td>delaunay_n19</td>
<td>132.46</td>
<td>111.78</td>
<td>111.78</td>
<td>101.59</td>
</tr>
<tr>
<td>poisson3Db</td>
<td>157.94</td>
<td>86.00</td>
<td>86.00</td>
<td>112.79</td>
</tr>
<tr>
<td>citationCiteser</td>
<td>135.59</td>
<td>70.33</td>
<td>135.59</td>
<td>134.23</td>
</tr>
<tr>
<td>FullChip</td>
<td>4081.38</td>
<td>3028.05</td>
<td>3028.05</td>
<td>3863.76</td>
</tr>
<tr>
<td>belgium_osm</td>
<td>180.88</td>
<td>206.83</td>
<td>180.88</td>
<td>240.03</td>
</tr>
<tr>
<td>com-YouTube</td>
<td>911.14</td>
<td>286.34</td>
<td>286.34</td>
<td>392.93</td>
</tr>
<tr>
<td>bcsstk13</td>
<td>2.90</td>
<td>1.82</td>
<td>1.82</td>
<td>0.62</td>
</tr>
<tr>
<td>bundle_adj</td>
<td>4395.73</td>
<td>437.39</td>
<td>437.39</td>
<td>840.43</td>
</tr>
<tr>
<td>SiO2</td>
<td>450.68</td>
<td>174.84</td>
<td>450.68</td>
<td>263.09</td>
</tr>
</tbody>
</table>

Table 3: Runtime of SpMM for the static and the dynamic scheduling in the left part of the table and the runtime of transfer tuning and Intel MKL in the right part of the table.

Figure 10: t-SNE plot of the SpMM embeddings for the sparse weights of a BERT model [39]. The embeddings are colored by the optimal scheduling type, i.e., static (purple ●) and dynamic (orange ●).
Transfer Tuning. Martins et al. [44] cluster C functions based on static features to select the optimal compiler passes according to the cluster assignment. Gibson and Cano [30] provide a constrained definition of the term transfer tuning as the reuse of optimizations found by auto-schedulers for specific operations in tensor programs. The discovered optimizations are matched by hand-crafted heuristics to other operations. Our approach extends this concept to intermediate representations and optimizations based on a fuzzy matching of node embeddings. The similarity of performance embeddings thereby generalizes hand-crafted transfer rules.

8 DISCUSSION

The following section briefly discusses possible extensions of the presented similarity-based framework.

Scalability. The density of the optimization database is a crucial hyperparameter for the validity of the similarity-based approach. However, the separation of the model and the transformations enables offline search for further optimizations. This allows to continuously improve the quality of the search by extending the database (i.e., online learning) with suboptimal examples. For existing auto-schedulers, a corresponding extension of the approach means expensive re-training and a significant increase in the scheduling space for all applications. This is a practical problem since current auto-schedulers often fail for basic applications, such as the jacobi2d benchmark on the model of Baghdadi et al. [5] or the max filter on Adams et al. [1]. A possible next step for the approach is to evaluate transfer tuning with larger databases.

Transformation Alignment. The matching algorithm matches a transformation to a parallel loop nest using the Hungarian method. However, the matching of a sequence of transformations is modeled greedily, which means that a database is required that covers symmetric cases as separate entries. However, such cases typically require a simple modification of the transformation sequence. For instance, a loop interchange, which is a common infix in transformation sequences, may often be skipped or replaced by a similar interchange for specific pairs of loop nests. This problem could be modeled as a sequence alignment problem, where the skipping or insertion of specific transformations are latent decisions (represented by, e.g., a Hidden Markov Model). Sequence alignments are well-known in the field of machine translation [46, 59]. Understanding performance optimization as a sequence alignment between a reference optimization and a similar loop nest gives rise to the idea of a model-based alternative to the model-free reinforcement learning approach presented by Steiner et al. [52].

Loop Fusion. The fusion of parallel loop nests is an important optimization to reduce the volume of necessary data movement. In order to support this optimization in the similarity-based framework, a model is necessary which produces subgraph embeddings for graphs of parallel loop nests. Such models are subject to current research [2].

Target Architecture. The separation of the model and the optimizations also facilitates porting the approach to new architectures. In particular, learning a representation for similarity search is significantly simpler than training a model that accurately predicts the speedups of complex optimization sequences. In fact, the dynamic encoding and the targets only need to be substituted by appropriate performance counters and metrics for the new target architecture. Performance models usually provide a good basis for finding relevant metrics and are available for most architectures, e.g., NUMA nodes [24], FPGA [22], GPU [45], and distributed computing [20].

9 CONCLUSION

In this paper, we present a similarity-based tuning framework that lifts peephole optimizations by fuzzy-matching larger program transformations. The approach separates the performance model from the optimizations in the form of performance embeddings and an optimization database. This enables local search for optimizations over the nearest neighbors in the embedding space.

We demonstrate the approach in different case studies highlighting the reduction of the search complexity by up to four orders of magnitude, and the extensibility of the approach to tailored optimizations on data-dependent applications, outperforming the state-of-the-art MKL library in certain use cases. The approach creates a new space that can be used for explainable and robust optimization, while remaining adaptive to future applications and hardware — transferring a new optimization technique is as simple as adding a row to the database.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory. LLNL-CONF-848643. This work received EuroHPC-JU funding with support from the European Union’s Horizon 2020 program and from the European Research Council under grant agreement PSAP, number 101002047, and grant DEEP-SEA, No. 955606. The authors also wish to acknowledge the support from the PASC program (Platform for Advanced Scientific Computing) for the DaCeMl project. T.B.N. (while at ETH Zurich) and P.S. were supported by the Swiss National Science Foundation (Ambizione Project #185778). L.T. wants to thank Hannah, Aileen, and friends for their support.

REFERENCES

A APPENDIX

The static encoding maps nodes and edges of an SDFG to a set of features. The mapping of SDFG node types to features is summarized in Table 4.

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Node</td>
<td>data type, bytes per element, shape, total size, stride, alignment, offset, transient, storage type</td>
</tr>
<tr>
<td>Map Entry</td>
<td>map level, map dimensions, map extents, map steps</td>
</tr>
<tr>
<td>Map Exit</td>
<td>one-hot encoding</td>
</tr>
<tr>
<td>Memlet</td>
<td>start access matrix, stop access matrix, steps vector, dynamic, indirection, reduction, type of reduction</td>
</tr>
</tbody>
</table>

Table 4: An overview of the static features selected for the static encoding of parallel loop nests. Most features directly correspond to the properties of nodes in an SDFG.

The dynamic encoding maps the profiling to 19 performance counters selected from 8 different groups. Table 5 lists the counters and groups in detail.

<table>
<thead>
<tr>
<th>Group</th>
<th>Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>INSTR_RETIRED_ANY</td>
</tr>
<tr>
<td>FP 32</td>
<td>FP_ARITH_INST_RETIRED_SCALAR_SINGLE</td>
</tr>
<tr>
<td></td>
<td>FP_ARITH_INST_RETIRED_128B_PACKED_SINGLE</td>
</tr>
<tr>
<td></td>
<td>FP_ARITH_INST_RETIRED_256B_PACKED_SINGLE</td>
</tr>
<tr>
<td></td>
<td>FP_ARITH_INST_RETIRED_512B_PACKED_SINGLE</td>
</tr>
<tr>
<td>FP 64</td>
<td>FP_ARITH_INST_RETIRED_SCALAR_DOUBLE</td>
</tr>
<tr>
<td></td>
<td>FP_ARITH_INST_RETIRED_128B_PACKED_DOUBLE</td>
</tr>
<tr>
<td></td>
<td>FP_ARITH_INST_RETIRED_256B_PACKED_DOUBLE</td>
</tr>
<tr>
<td></td>
<td>FP_ARITH_INST_RETIRED_512B_PACKED_DOUBLE</td>
</tr>
<tr>
<td>Branching</td>
<td>BR_INST_RETIRED_ALL_BRANCHES</td>
</tr>
<tr>
<td>DRAM Controller</td>
<td>MEM_INST_RETIRED_ALL_LOADS</td>
</tr>
<tr>
<td></td>
<td>MEM_INST_RETIRED_ALL_STORES</td>
</tr>
<tr>
<td>Main Memory</td>
<td>CAS_COUNT_RD</td>
</tr>
<tr>
<td></td>
<td>CAS_COUNT_WR</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>L2_LINES_IN_ALL</td>
</tr>
<tr>
<td></td>
<td>L2_TRANS_L2_WB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>L1D_REPLACEMENT</td>
</tr>
<tr>
<td></td>
<td>L1D_M_EVICT</td>
</tr>
</tbody>
</table>

Table 5: An overview of the performance counters selected for the dynamic encoding on the Intel Xeon Gold 6140 CPU.