Pattern-based Autotuning of OpenMP Loops using Graph Neural Networks

Akash Dutta*, Jordi Alcaraz†, Ali TehraniJamsaz†, Anna Sikora‡, Eduardo Cesar‡ and Ali Jannesari∗

*Department of Computer Science
Iowa State University, Ames, Iowa, USA
Email: {adutta, tehrani, jannesari}@iastate.edu
†OACISS, University of Oregon, Eugene, USA
Email: jordia@uoregon.edu
‡CAOS Department, Universitat Autonoma de Barcelona, Spain
Email: {anna.sikora, eduardo.cesar}@uab.cat

Abstract—Stagnation of Moore’s law has led to the increased adoption of parallel programming for enhancing performance of scientific applications. Frequently occurring code and design patterns in scientific applications are often used for transforming serial code to parallel. But, identifying these patterns is not easy. To this end, we propose using Graph Neural Networks for modeling code flow graphs to identify patterns in such parallel code. Additionally, identifying the runtime parameters for best performing parallel code is also challenging. We propose a pattern-guided deep learning based tuning approach, to help identify the best runtime parameters for OpenMP loops. Overall, we aim to identify commonly occurring patterns in parallel loops and use these patterns to guide auto-tuning efforts. We validate our hypothesis on 20 different applications from Polybench, and STREAM benchmark suites. This deep learning-based approach can identify the considered patterns with an overall accuracy of 91%. We validate the usefulness of using patterns for auto-tuning on the number of threads, scheduling policies and chunk size on a single socket system, and the thread count and affinity on a multi-socket machine. Our approach achieves geometric mean speedups of 1.1× and 4.7× respectively over default OpenMP configurations, compared to brute-force speedups of 1.27× and 4.93× respectively.

Index Terms—OpenMP, Pattern detection, Auto-tuning, GNN

I. INTRODUCTION

OpenMP, due to its short learning curve and ease of use, is widely used for porting serial scientific applications to parallel ones to take advantage of multi and many-core processing. However, simply annotating serial code with OpenMP pragmas is not enough to achieve peak performance. There are multiple tuning parameters that can be tweaked to influence the performance of OpenMP code. Additionally, such applications are usually designed to accept varied inputs. Manually identifying profitable configurations for a number of applications and their associated variable inputs can translate to a time consuming ad-hoc approach. Exhaustively exploring a search space to identify the best configurations is another option. Although this might work well for smaller search spaces, such an approach is infeasible for larger search spaces. Thus, a majority of the works in this field employ heuristic or rule-based approaches to expedite this process. However, these approaches tend to execute code multiple times before converging at a solution. To this end, we propose a graph neural network (GNN) based pattern guided approach to identify the best OpenMP runtime configurations. Keutzer et al. in [1] elaborated on the need of design patterns to develop high performing parallel code. It has also been previously reported that using design patterns to annotate OpenMP code regions lead to performance improvements [2], [3]. Most existing code pattern detection techniques depend on expert designed heuristics and rules [2], [4]. In this paper, we focus on loop-level parallelism, which are a sub-part of the Implementation Strategy Patterns described in OPL [1], and present this as a building block for expanding our pattern detection to other patterns described in OPL.

Using machine learning to identify code patterns does not limit the identification and analysis to pre-defined patterns or models. Without expert knowledge, the impact of different inputs and configurations on the performance of parallel code is difficult to predict. A deep learning method, however, because of its probabilistic and adaptive nature, can roughly approximate these trends. To analyze program structure and behavior, we model static code features as flow multi-graphs, and dynamic features by performance counters. Flow graphs help to capture the structural features of source code, and the dependencies in source code. Performance counters are also essential to model the execution behavior of programs, which static code graphs cannot do. We use such a characterization to build a model that can accurately predict parallel loop patterns and we extend this approach to develop a pattern guided tuning method. Our tuning approach is not application specific and can easily be repurposed for various tuning tasks giving users flexibility of use.

To summarize, the contributions of this paper are as follows:

- Developing a GNN-based pipeline to predict patterns in parallel loops with an mean accuracy of 91%.
- Designing models to show the importance of code patterns in DL-based tuning, achieving mean speedups of 1.1× over default OpenMP runtime configurations.
- Extending our tuning approach to multi-socket NUMA systems and achieving mean speedups of 4.7× over default runtime configurations.
II. BACKGROUND AND RELATED WORKS

In this section, we briefly describe the ideas and concepts relevant to this work.

A. Static Code Representations for Deep Learning

Representation learning is being increasingly used for code analysis and optimization tasks [5]. However, its use necessitates the use of a strong code representation capable of capturing the inherent features in source code. A majority of such code representations looks at code as a sequence of lexical tokens [6]. This fails to capture the structured nature of programs. To overcome this, representations capturing syntactic and semantic features been proposed [5], [7], [8]. But these methods, often, do not take into account control, data, and call flows existing in a program. PROGRAML [6] is such a tool that represents the semantic and structural features of code in a flow-aware multi-graph. We use PROGRAML to transform parallel loops to their corresponding graphs.

B. Patterns in Parallel Code

Certain code sections in parallel programs belong to categories of highly recurring patterns for which efficient and well-known parallelization solutions exist. A wide range of parallel patterns exist such as loop-level parallelism, task level parallelism, etc. In this paper we only focus on loop-level parallelism, specifically do-all, reduction and stencil patterns.

Do-All Pattern. Do-all loops do not have any loop carried dependencies. In a do-all loop, different memory locations are accessed in each iteration. This makes these loops easily parallelizable and suitable for multi-core programming.

Reduction Pattern. A reduction loop allows certain dependencies to exist in reduction variables. Reduction variables can be used to combine multiple elements of an array (e.g. to sum elements of an array). Loops having this pattern are easily parallelized as the operands in these cases are associative and commutative. However, in contrast to do-all loops, reduction loops are parallelized in a tree structure where nodes of the tree are used to hold results of intermediate computations, which are iteratively combined to arrive at the final result.

Stencil Pattern. A stencil pattern is characterized as a loop that accesses not only a single element but also its “neighbors”. The value of an element is updated by using values of neighboring elements in a fixed pattern. Usually, boundary values in such a structure remain unchanged and influence the value updations in its neighbors.

Several works have suggested identifying and using patterns in code for a better and more uniform way of writing parallel code and improving performance [2], [3], [9]–[11].

C. Graph Neural Networks

DL has revolutionized the application of machine learning in tasks that generate data represented in the Euclidean space. However, there exist an increasing number of applications where data is generated from non-Euclidean space [12]. The relations and dependencies between objects in such data can more readily be represented as a graph. GNNs were proposed as a means of modeling such data. Graph Convolutional Networks (GCNs) are a form of GNNs aimed at generalizing the common sliding window convolution operation on grid data in regular Convolutional Neural Networks to graphs. We use Relational Graph Convolutional Network (RGCN), a variation of GCN, to model our program graphs. RGCNs were proposed to enable networks to better model large-scale relational data [13]. Unlike GCNs, RGCNs work with relation specific transformations annotated by the type and direction of edges. We use RGCNs to model our flow-aware code graphs.

D. Performance Tuning

OpenMP exposes a number of configurations for runtime optimization of code. For scientific applications, tuning these knobs can impact code performance. A large body of research exists on tuning runtime parameters or configurations for parallel code. Sreenivasan et al. in [14] described a lightweight model-based auto-tuning framework for tuning OpenMP code. OpenTuner [15] is an extensible framework for domain-specific tuning that uses ensembles of search techniques for search-space exploration. An alternative to classical auto-tuning is to use machine learning based approaches. To this end, [16], [17] propose machine learning based approaches to OpenMP autotuning. Artemis [18] and ytopt [19] are two examples of ML-based parameter tuning frameworks. Recently, deep learning is being more regularly used for code analysis and optimizations. To this end, many code representations have been used [6], [7], [20]–[23], which have been used to good effect for several optimization tasks such as heterogeneous device mapping, algorithm classification, thread coarsening factor to name a few.

III. METHODOLOGY

A challenge with any deep learning approach for code analysis or auto-tuning lies in identifying an optimum set of features. Program properties and structure, along with its execution behavior plays an important role in identifying tuning parameters. We represent code properties (static features) as structure preserving, flow-aware multi-graphs, and execution behavior (dynamic features) with performance counters. Static or dynamic features on their own cannot link the impact of code structure and dependencies on hardware components. Both are needed to analyze how different code structures and patterns influence code execution.

We use RGCNs to model our code graphs. The high-level embeddings obtained from this modeling is concatenated with a reduced set of performance features to identify code patterns, and for downstream tuning tasks. An overview of our approach can be seen in Figure 1 and each step is discussed in the following paragraphs.

A. Step A: Static Code Representation

We initially compile the parallel code to their intermediate representation (IRs) in all our benchmark files. These IRs contain the OpenMP code regions as outlined functions. We use the llvm-extract tool to extract these outlined parallel
The effectiveness of this approach is shown in Section IV.  

1) OpenMP Thread and Scheduling Optimizations on non-NUMA architecture: To validate the effectiveness of our tuning approach, we first analyze our model’s performance, on optimizing the thread count and, OpenMP scheduling and chunk size on a single socket system. Often, using the default configuration of all threads, static scheduling, and dynamically calculated chunk sizes does not produce the best performance. Tuning these parameters often leads to improved load balancing and worksharing in OpenMP loops. This influences code execution and can potentially improve performance. Optimizing these parameters can lead to reduced cache misses and branch mispredictions which in turn can lead to faster code executions. Additionally, one of the goals of this experiment was to create a search space of these
parameters, that is at the same time, manageable, but also large enough to afford us key insight about the impact of these parameters on the behavior of a parallel loop. To this end we define the configuration search space as shown in Table I. We explore all possible combinations of these parameters to obtain performance counters and, identify the best configuration out of all possible 168 configurations for each code region and input size. This serves as the dataset for this experiment, and is also used as an oracle to evaluate our model’s predictions.

Most existing execution based auto-tuners, when faced with such a large search space, would iterate a large number of times before identifying profitable tuning parameters. However, during inference, our approach only requires only two executions (the selected PAPI events are incompatible and cannot be profiled in a single run) to collect performance counters of an unseen OpenMP loop.

<table>
<thead>
<tr>
<th>Search Space (Skylake)</th>
<th>Parameter Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of threads</td>
<td>1, 2, 3, 4, 5, 6, 7, 8</td>
</tr>
<tr>
<td>Scheduling Policy</td>
<td>STATIC, DYNAMIC, GUIDED</td>
</tr>
<tr>
<td>Chunk Sizes</td>
<td>1, 8, 32, 64, 128, 256, 512</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Search Space (SandyBridge)</th>
<th>Parameter Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of threads</td>
<td>1, 2, 3, ..., 32</td>
</tr>
<tr>
<td>Thread Affinity</td>
<td>close, spread</td>
</tr>
</tbody>
</table>

2) OpenMP Thread and Affinity Optimizations on NUMA architectures: To verify the generalizability of our tuning approach, we also experiment with a different tuning task on a different architecture. We test the validity of our approach to tuning tasks on multi-socket NUMA systems by tuning the number of threads and the thread affinity. On NUMA architectures, allocation of local memory to processors can improve CPU performance. The close association of local memory and processors, however, increases data latency when other processors try and access data from non-local memory. This is often a problem for multicore programming as different threads trying to access data from non-local memory might end up slowing down executions. Therefore, optimizing such locality issues along with the number of threads, has a big impact on parallel code execution on NUMA machines.

Thread affinity is used to determine the mapping of threads to cores. OpenMP provides two ways of performing this: close affinity, that assigns threads to contiguous cores, and spread affinity, that assigns threads to cores in a round-robin fashion. The assignment of threads to cores has a big impact on memory footprint, and thus, performance. Appropriate placement of threads and associated data on cores directly impact code performance. Parameters that reduce data movement overhead between cores can improve data latency issues, leading to improved performance.

For this experiment, we defined a search space (Table I) that considers 32 different parameter values of thread counts, and two values for thread affinity, creating a search space with 64 possible configurations. As before, in the inference step, we only need to profile a code region and an input a few times before feeding it into our DL-model. The model setup and inputs are similar to the models described previously and shown in Figure 1.

IV. EXPERIMENTS

We use the DiscoPoP [2] tool and manual inspection to identify the loop patterns in the code regions, and use them as the labels for the pattern detection task. In order to identify the best tuning parameters, we explore all combinations of input sizes and tuning parameters for all OpenMP loops. We use this exhaustive exploration as an oracle to identify the best tuning parameter values. The identified values are then used as labels while training our models.

A. Experimental Setup

For our experiments, we use two systems: a multi-socket system with Intel Xeon E5-4620 processor (Sandy Bridge μ-architecture), with 4 sockets and 8 cores per socket, and a single-socket system with Intel Xeon Silver processor (Skylake μ-architecture) with 1 socket and 8 cores. The data collected on the multi-socket system is used for the experiments in Section IV-E2), whereas the data from the single socket system is used for the experiments in Section IV-E1. We build our DL models using Pytorch and Pytorch Geometric.

B. Selecting the benchmarks

All experimentation in this study was performed on benchmark applications from the Polybench and STREAM suites. The applications in this benchmark suite cover a wide variety of computational kernels and workloads used frequently in scientific applications and can be easily instrumented with varying input sizes. Additionally, the presence of one primary kernel in each application eases the task of identifying the bottleneck code regions and aids the analysis of the effects of code structure on performance.

C. Data Collection and Preprocessing

We use 25 different input sizes ranging from 3.5KB to 0.5GB to characterize the behavior of OpenMP loops across different inputs. The inputs are selected such that these stress each cache level to different degrees. The smallest input sizes are deliberately designed to fit within the first-level cache (L1), whereas the largest ones are set up to overflow the last-level cache (L3). The intermediate ones exercise the caches to varying degrees. The instrumented code is then profiled 100 times for each input size and configuration to collect performance counters, leading to more than 1 million data points. We then use the mean of the performance counter values, and use correlation analysis to identify five counters most highly correlated to execution time. The identified performance counters for the experiments in Section IV-E1 are L1 store and cache misses, L2 cache misses, L3 cache writes, and the number of cycles with no instruction issues. For the experiments in Section IV-E2, the identified performance counters are the L3 cache misses, L2 cache hits and misses, L1 load misses and the total number of instructions. The pattern detection experiments need to be performed on data from only one machine as code patterns are machine independent. We
use the data collected on the single-socket Skylake machine for classifying the patterns.

D. Pattern Detection

In this section, we aim to identify parallel loop patterns using graph neural networks. For the purposes of this work, we limit ourselves to identifying three very common loop patterns; do-all, reduction, and stencil. Our pattern classification model expects two primary inputs: i) graph representations of the OpenMP loops, and ii) performance counters to represent execution behavior. The graph representation of code region is obtained through the PROGRAML tool. An RGCN model is used to model these code graphs. Modeling only the OpenMP loops as graphs was an optimization considered in this work. The code graphs of entire applications are larger and computationally more expensive to model than these loop graphs, without significant difference in performance. The code graph modeling consists of an embedding layer that acts as a look up table to map code text to an index, and layers of RGCN that takes as input a code graph, and performs weight updates for graph nodes based on adjacent node features and associated edge features. The outputs from the RGCN layers are then concatenated with the performance counters and fed into the FCNN layers. Prior to concatenation, the performance counters are normalized with respect to the number of reference clock cycles and scaled to a [0,1] range. This network is expected to classify each loop to its correct pattern. As hyperparameters, we consider tanh, ReLU, and sigmoid as activation functions, and Stochastic Gradient Descent, Adam, and AdamW as optimizers. We find that for these experiments, ReLU and AdamW lead to smoother convergence and produces better results.

We randomly select 50% of the loops for each code pattern for training, and the remaining data is used for validation. With this approach, we observe that our model can identify do-all loops with an average accuracy of 100%, reductions with an average accuracy of 83%, and stencils with a mean accuracy of 90%. Overall, our model identifies the patterns with an accuracy of 91%.

E. Pattern-based Auto-Tuning

Our aim in this section is to identify the influence of loop patterns on auto-tuning tasks. To this end we explore the effectiveness of our approach on two relevant tuning tasks across single-socket and multi-socket systems.

1) Optimizing the number of threads, scheduling policy and chunk size on single-socket architecture: OpenMP provides a number of configurable runtime and environmental parameters that can be modified to influence code execution. The most commonly used of these parameters are thread count, and scheduling policy and chunk size. The OpenMP loops graphs and performance counters are modeled using the RGCN and FCNN layers as outlined in Section IV. This model is trained to predict the best configuration for the single socket Skylake machine used in this experiment.

For validation, we perform a randomized 80/20 split of the sets of kernels belonging to each pattern into the training and validation set. This ensures that both the training and validation set includes data from kernels belonging to each pattern. Along with the code graphs, the training and validation data consists of the five normalized performance counters identified in Section IV-C for each input size, and the appropriate loop pattern. During inference, our model is expected to predict the best configuration for each loop at each input size. We only need to feed a single set of performance counters (this requires two executions), along with the code graphs, patterns, and input size for prediction.

The results show that such a pattern based approach improves results compared to a non-pattern based approach. This pattern-based tuning leads to speedups of 1.1× over default configurations using all threads (8), static scheduling and dynamically calculated chunk sizes. If patterns are not considered, the normalized speedups fall to 0.93×. The best achievable speedups for this experiment on the given machine is 1.27× over default configurations. The results for all scenarios are shown in Table II.

2) Optimizing the number of threads and affinity on multi-socket NUMA architecture: On NUMA architectures, along with identifying the best thread count, the assignment of threads to cores has a significant impact on performance and memory footprint. OpenMP provides two mechanisms of assigning threads to cores: close and spread.

The goal of this experiment is to identify the best combination of any of the two thread affinities and any one of the 32 possible thread counts for the system under test. Our deep learning model, as before, consists of two parts: a set of RGCN layers for modeling the code graphs, followed by fully connected layers that work as a classifier for identifying the best configurations. The RGCN layers expect as input the code graphs obtained in previous steps. The output from the RGCN layers are concatenated with five performance counters, identified in Section IV-C, the appropriate code pattern, and the associated input size for each set of performance counters.

As in the previous experiment, the training and validation sets are obtained by performing an 80/20 split on the set of kernels associated with each pattern. The only changes lie in the final prediction it must make. As in the experiments in Section IV-E1, we only need to feed a single set of performance counters, code graph, and loop pattern for a target loop and input size. Our pattern-based tuning approach outperforms a tuning approach that does not consider patterns and leads to normalized speedups of 4.7× over default runtime.
configuration of using all threads (32) and close thread affinity. The GNN based approach that does not use patterns leads to speedups of 3.35×. The best achievable speedups for this experiment is 4.93×. The results of this experiment are shown in Table II.

V. CONCLUSION AND FUTURE WORKS

Through this work, we have presented a preliminary work on using GNNs for identifying loop patterns and using these patterns for guiding tuning efforts of OpenMP loops. Because of the exploratory nature of this work, only simple loop patterns (do-all, reduction, and stencil) have been considered. A tool like DiscoPoP can easily identify do-all and reduction patterns, however it cannot identify stencils. A large number of scientific parallel applications implement these patterns. Therefore, identifying and tuning such loops is essential.

Our GNN-based approach achieves good results in identifying loop patterns and tuning them. As with all deep learning approaches, training is an overhead. However, modeling only loop graphs instead of applications, and using a reduced number of performance counters speeds up training. Additionally, our approach suffers from a profiling overhead. However, the number of executions during inference is only two. In comparison, brute force tuning would execute code 168 and 64 times for the corresponding tuning experiments outlined in this paper. We aim to build upon this work to include more complex patterns based on the OPL standard. Several other patterns at varying granularity will be considered to improve coverage of our pattern detection.

ACKNOWLEDGEMENTS

We would like to thank the Research IT team1 of Iowa State University for their continuous support in providing access to HPC clusters for conducting this research. This research has also been supported by Ministerio de Ciencia e Innovación MCIN AEI/10.13039/501100011033 under contract PID2020-113614RB-C21 and by the Generalitat de Catalunya GenCat-MCIN AEI/10.13039/501100011033 under contract PID2020-

REFERENCES


1https://researchit.las.iastate.edu


