# Performance Optimization using Multimodal Modeling and Heterogeneous GNN

Akash Dutta
Department of Computer Science
Iowa State University
Ames, IA, USA
adutta@iastate.edu

Jordi Alcaraz
OACISS
University of Oregon
Eugene, Oregon, USA
jordia@uoregon.edu

Ali TehraniJamsaz
Department of Computer Science
Iowa State University
Ames, IA, USA
tehrani@iastate.edu

Eduardo Cesar
CAOS Department
Universitat Autònoma de Barcelona
Barcelona, Spain
Eduardo.Cesar@uab.cat

Anna Sikora
CAOS Department
Universitat Autònoma de Barcelona
Barcelona, Spain
Anna.Sikora@uab.cat

Ali Jannesari
Department of Computer Science
Iowa State University
Ames, IA, USA
jannesari@iastate.edu

## ABSTRACT

Growing heterogeneity and configurability in HPC architectures has made auto-tuning applications and runtime parameters on these systems very complex. Users are presented with a multitude of options to configure parameters. In addition to application specific solutions, a common approach is to use general purpose search strategies, which often might not identify the best configurations or their time to convergence is a significant barrier. There is, thus, a need for a general purpose and efficient tuning approach that can be easily scaled and adapted to various tuning tasks. We propose a technique for tuning parallel code regions that is general enough to be adapted to multiple tasks. In this paper, we analyze IR-based programming models to make task-specific performance optimizations. To this end, we propose the *M*ultimodal *G*raph Neural Network and *A*utoencoder (MGA) tuner, a multimodal deep learning based approach that adapts Heterogeneous Graph Neural Networks and Denoizing Autoencoders for modeling IR-based code representations that serve as separate modalities. This approach is used as part of our pipeline to model a syntax, semantics, and structure-aware IR-based code representation for tuning parallel code regions/kernels. We extensively experiment on OpenMP and OpenCL code regions/kernels obtained from PolyBench, Rodinia, STREAM, DataRaceBench, AMD SDK, NPB, NVIDIA SDK, Parboil, SHOC, LULESH, XSBench, RSBench, miniFE, miniAMR, and Quicksilver benchmarks and applications. We apply our multimodal learning techniques to the tasks of (i) optimizing the number of threads, scheduling policy and chunk size in OpenMP loops and, (ii) identifying the best device for heterogeneous device mapping of OpenCL kernels. Our experiments show that this multimodal learning based approach outperforms the state-of-the-art in almost all experiments.

*HPDC '23, June 16–23, 2023, Orlando, FL, USA*
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0155-9/23/06.
https://doi.org/10.1145/3588195.3592984

## CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; **Machine learning**.

## KEYWORDS

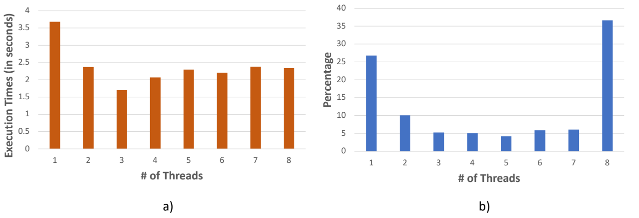
Auto-tuning, Multimodal learning, Heterogeneous Graph Neural Networks, OpenMP, OpenCL

**ACM Reference Format:**
Akash Dutta, Jordi Alcaraz, Ali TehraniJamsaz, Eduardo Cesar, Anna Sikora, and Ali Jannesari. 2023. Performance Optimization using Multimodal Modeling and Heterogeneous GNN . In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3588195.3592984

## 1 INTRODUCTION

With the onset of the exascale computing era, a lot of attention is now focused on HPC landscapes. However, the benefits of parallel programming is not just limited to supercomputers. Most systems nowadays have multi/many-core architectures. These hardware capabilities have led to the increased adoption of parallel programming models such as OpenMP and OpenCL for writing parallel code. Their shorter learning curves and ease of use has led to such programming models being used extensively not just for CPU programming, but also for programming accelerators such as GPUs. Although these programming models have made it easier to convert serial code to parallel, they do provide users and programmers with various parameters that can be tweaked to highly impact performance. However, selecting these parameters is often cumbersome and often needs expert guidance. We aim to help address this by proposing a deep learning based IR-modeling technique for faster convergence and better results compared to state-of-the-art tools.

***Motivation.*** As an example, we evaluate the execution time of the OpenMP version of the kmeans kernel from the Rodinia [16, 17] benchmark suite at different thread counts on an eight core system. We see significant difference in performance by varying the number of threads (Figure 1a). There are four thread counts that achieve better performance than the default eight threads, improving execution time by upto 27%. kmeans, like many others, allows variable user inputs. Repeating such a brute-force approach for a large set of



a)

b)

**Figure 1: a) Execution times of** `kmeans` **benchmark with different threads. b) Distribution of best thread counts across all** `OpenMP` **loops and inputs in the dataset**

applications and variable inputs is not feasible. The full extent of the tuning task at hand can be shown through Figure 1b, where across 45 OpenMP loops, and 30 different inputs, approximately 64% combinations require tuning to identify the best thread count. Larger search spaces would render such a brute-force approach unrealistic. Our process aims to ease this while achieving better results than existing auto-tuners.

***Prior Works.*** A majority of the performance optimization works follow either static or dynamic analysis based approaches. Static analysis based works regularly target compiler optimizations which are a set of carefully handwritten heuristics [81]. Several works have used machine learning algorithms to improve the optimization decisions made by compilers [31, 33, 45, 51, 55, 70].

Dynamic analysis based autotuners have to extensively execute source code and are usually search based tools [8, 74, 76]. Recently, several tools have employed Bayesian optimization (BO) based surrogate models for tuning purposes such as [9, 56, 64] to reduce the execution overhead of autotuners. However, these still need a number of evaluations for each combination of target application and input. Overall, this can become quite expensive, when inputs to code kernels vary with great regularity. Similar to BO, a deep learning (DL) based approach, can aggressively prune non-beneficial points through learning. However, such DL models are usually not application and (or) input dependent; thus are more general-purpose in nature and almost always require minimal sampling and evaluation during inference, making them good candidates for deployment in real-world settings.

Several approaches using DL have been proposed for performance optimization tasks. Most of these propose a new method of representing code to achieve high-quality results [15, 20, 81]. These approaches, however, only consider task specific features inherent to one form of code representation. This exposes the shortcomings of each representation and leads to a loss of some syntactic, semantic, and structural characteristics of code. In contrast, we propose combining more than one such representation, to use their individual strengths to overcome the shortcomings of the other. To this end, we propose a code modeling technique that builds on existing representations and improves results by adapting multimodal learning to the task of code modeling.

***Our contributions.*** We propose the MGA tuner, that models two dissimilar static code representations as separate characterizations of the same piece of code. This allows the conjunctive modeling of multiple code representations targeted towards a common end goal. To this end, we propose modeling a distributed program vector and a graphical code embedding as different modalities of our multimodal learner. This can address the shortcomings of other unimodal approaches. In this learner, the code graphs will be modeled by a heterogeneous graph neural network, and the distributed vectors are modeled using a denoising autoencoder. Moreover, as static features themselves cannot model the execution behavior with multiple inputs, we augment these with performance counters (dynamic features). Similar to some of the approaches discussed in the previous paragraph, our modeling technique is intermediate represenatation (IR)-based, making our code modeling language and architecture agnostic. We will show later that for tuning OpenMP runtime parameters, our approach produces better results than state-of-the-art autotuners, while needing less executions. We will also show that our approach outperforms existing techniques on OpenCL-based heterogeneous device mapping tasks. To summarize, our contributions are as follows:

- Designing a new IR based hardware-independent multimodal code modeling technique that encapsulates syntactic, semantic, and structural code features.
- Developing heterogeneous graph neural network models for modeling flow graphs.
- Using denoising autoencoders for modeling distributed code vectors
- Designing a DL-based tuning approach for OpenMP runtime parameters with geometric mean performance gains of 3.4× while predicting OpenMP threads and 2.23× for predicting threads, schedule, and chunk size.
- Quantifying the impact of performance counters on DL-based performance tuning.
- Analyzing the $\mu$-architecture portability of our approach.
- Building a multimodal learner for the task of OpenCL based heterogeneous device mapping achieving state-of-the-art accuracy of $\sim$ 98%

***Outline.*** Section 2 outlines the topics of interest for this paper, followed by our approach and experiments in Sections 3 and 4, respectively. We outline related works in Section 5, and discuss and conclude our paper in Sections 6 and 7.

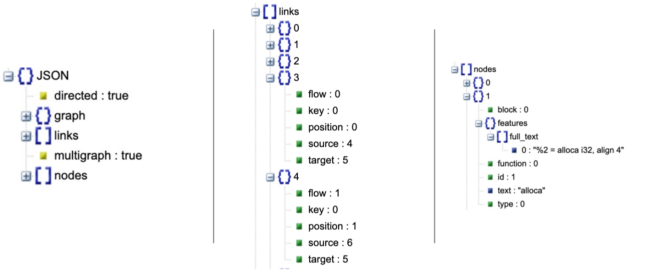
## 2 BACKGROUND AND OVERVIEW

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

### 2.1 Code Representations and Deep Learning

Representation learning is being increasingly used for code modeling tasks. A lot of previous works have represented programs as a sequence of lexical tokens [20]. However, this fails to capture program structure. To overcome this, syntax as well as semantics based representations have been proposed [4, 5, 15, 23, 63]. But these methods do not take into account control, data, and call flows in a program. Several approaches have been suggested to represent these flows [13, 47, 69]. However, these often lack the information provided by syntactic and structural modeling of source code.

IR2Vec [81] is a flow-aware code representation that is not structurally aware. It is a scalable encoding infrastructure that represents programs as a distributed embedding in continuous space.



```
{} JSON
    directed : true
  {} graph
  [] links
    multigraph : true
  [] nodes
```

```
[] links
  {} 0
  {} 1
  {} 2
  {} 3
      flow : 0
      key : 0
      position : 0
      source : 4
      target : 5
  {} 4
      flow : 1
      key : 0
      position : 1
      source : 6
      target : 5
```

```
[] nodes
  {} 0
  {} 1
      block : 0
    {} features
      [] full_text
          0 : "%2 = alloca i32, align 4"
      function : 0
      id : 1
      text : "alloca"
      type : 0
```

**Figure 2: Code graphs in JSON format. The left image shows overall graph structure. The middle figure shows edge features. The *flow* attribute denotes the type of program flow. The right image shows the node features.**

PROGRAML [20] is another IR-based code representation that can model code flow information along with the code structure as multi-graphs. Each multi-graph has a vertex for instruction and control-flow edges between them. Data flow is represented by including separate vertices for variables and constants and associated data-flow edges to instructions. Call flow is represented by edges between callee functions and caller instruction vertices. An example of such a graph used in this paper is shown in Figure 2.

For modeling static code features, we represent code as a multimodal problem that considers these two embedding techniques as separate modalities allowing us to overcome the shortcomings of each modality on its own. This multimodal approach allows our model to extract the syntactic, semantic, and structural features of code.

## 2.2 Performance Profiling

Static analysis is a powerful method for analyzing program properties. But, dynamic analysis is often essential for understanding the execution behavior of programs with various input sizes. Performance profiling is a means to this end. It is widely used to analyze how code/code section impacts the hardware components. Performance counters are used by developers to identify bottlenecks and scope of improvement in code execution. In this work, we use such counters to study the impact of various inputs on code execution. *perf*, Likwid [79], PAPI [57] are a few commonly used tools for profiling. We use PAPI to profile each loop in Section 4.1.

## 2.3 Graph Neural Networks

DL has revolutionized the application of machine learning in tasks that deal with data from Euclidean space. However, data is being increasingly generated from non-Euclidean space [86]. Such data can more readily be represented as a graph. Graph Neural Networks (GNNs) were proposed as a means of modeling such data. Almost all GNNs are implemented using Message Passing Neural Networks[29] (MPNN). The goal of these networks is to learn the latent space representation of each node through its neighbouring nodes in the graph. There are 3 main functions for constructing an MPNN: i) *Message:* constructs communication between neighboring nodes, ii) *Aggregate:* aggregates messages received from neighbouring nodes, iii) *Update:* updates the target node embedding according to Message and Aggregate functions.

Recent advances in GNNs have led to the proposal of **Heterogeneous Graph Neural Networks** [82]. Such models are used to accurately model diverse data with multiple relations. Real world graphical data usually consists of different sets of entities and relations and cannot be effectively modelled by homogeneous GNNs due to differences in node and edge features, and dimensionality. To overcome this issue, heterogeneous GNNs were proposed. In this work, we use heterogeneous GNNs to model our flow multi-graphs.

## 2.4 Autoencoders

Classic autoencoders are a type of deep learning model where the inputs and outputs are ideally the same. Most autoencoders follow the classic *encode-code-decode* setup. Given an input, the input is passed through layers of fully connected neural networks (ANN/MLP), called the encoder, which aims to compress the input to a smaller dimension. The encoder layers are followed by a code layer, which is usually a single MLP layer with a user-defined dimensionality (number of nodes). The code layer are followed by the decoder layers, which are nothing but layers of MLPs. The last layer of the decoder must have the same dimensionality as the input layer. Autoencoders are usually unsupervised techniques, where these models usually learn to approximate the identity function. Autoencoders are commonly used for feature selection and extraction.

**Denoising autoencoders (DAEs )** provide a twist on classic autoencoders where the inputs are selectively corrupted by randomly modifying certain inputs. The most common practice is to set a percentage of inputs to zero. The target in this case becomes the uncorrupted inputs. Because the training process for DAEs makes use of example/target pairs to gauge training quality, it becomes a self-supervised technique. The main task for DAEs thus is compression. In this work, we have used DAEs to model code vectors obtained through IR2Vec for feature extraction and compression.

## 2.5 Multimodal Deep Learning

Multimodal learning refers to relating information from multiple sources towards a common goal [59]. If there are multiple methods of modeling a target task, a problem can be assigned as multimodal, with each modeling technique defined as a unique modality. Multimodal learning has thus far mostly been applied to audio and video analysis, speech synthesis, and gesture recognition tasks [72]. For example, in image and video description tasks, the visual content and the associated textual description can be considered as different modalities with the same target – to enable the viewer to perceive the content and meaning of the image/video.

We take inspiration from these ideas and apply it to the task of code representation. A sequential and graphical code representation can represent different modalities of the same piece of code. The most common approach to multimodal modeling is to obtain high level embeddings from different sources and associate them towards a common task. Generally, early fusion and late fusion are two techniques used for associating data from disparate sources in multimodal learning [61]. On a high level, early fusion can be thought of as feature level fusion, where data from multiple sources are integrated into a single feature vector, before being used as input to a machine-learning model. Late or decision-level fusion refers to aggregating outputs from multiple models built on top of different modalities. This is often used as errors from multiple models are usually unrelated and such a method is feature independent. In this paper, we use late fusion for merging the outputs obtained by modeling two separate modalities.

## 3 THE MGA TUNER

This section presents a novel framework that adapts advanced deep learning (DL) techniques to performance tuning tasks. We argue that for DL-based code modeling, code syntax, semantics, and structure are extremely important for proper understanding of code. However, using a combined representation would make modeling them too complex, lead to increased feature overlap, reduced specificity in identifying relevant features, and introduction of noise and conflicts. To this end, we propose using two different code representations as separate modalities: i) a graphical code representation that can encode the code structure as a graph, ii) a distributed program vector representation that can encode syntactic and semantic features. A distributed vector representation can capture the relations within an instruction, but cannot effectively capture program structure. A graphical code representation, on the other hand, can fully capture code structure along with certain semantic features such as program flow. We aim to model the first modality using a heterogeneous graph neural network, and the second one using a denoising auto-encoder (DAE).

However, these static code features are not sufficient for modeling the execution behavior of code with different inputs. Therefore, we augment these features with dynamic features such as performance counters to include additional information about program behavior/setup with varied inputs. Figure 3 presents an overview of this tuning approach and Table 1 shows an overview of our model architecture.

### 3.1 Representing the Code

Our multimodal code modeling is built on top of two very different state-of-the-art code representations (IR2Vec [81] and PROGRAML [20]). Although the following part of the modeling process is done in parallel, we present these separately for improved readability and understanding.

To represent the first modality, each IR is passed through the PROGRAML tool to obtain the corresponding code graphs, as shown in the upper half of Figure 4. Along with representing the code structure, these code graphs also capture the data flow, control flow, and call flow in a single unified multi-graph, as shown in Figure 2. To represent the second modality, the code region/loop IRs are used to generate a seed embedding vocabulary with the IR2Vec encodings, as shown in the lower half of Figure 4. This seed embedding is then used to obtain the code vectors for each kernel used in the experiments. These code vectors and graphs are then passed through the code and performance modeling step as outlined in Section 3.2.

### 3.2 Performance Modeling

The kernel IRs in our dataset are first transformed to a form usable by DL networks using techniques discussed in Section 3.1. Experiment specific features are also used to augment the static feature set. In case of the OpenMP experiments, performance counters are collected and used to incorporate the impact of various inputs on code execution. For the experiments on OpenCL kernels, we have used transfer and workgroup sizes as additional input features to our models. These are discussed in further detail in Section 4. These, along with the static features, form the inputs to our models. As shown in Figure 3, our multimodal learning-based performance model can be abstracted into 3 high-level parts:

***Heterogeneous GNN modeling of flow graphs.*** As mentioned in Section 2.3, GNNs have been used for modeling graphical data. Heterogeneous GNNs have been successfully used for modelling real-world datasets with diverse node and edge attributes. The code graphs used in this paper encapsulate the flow information in programs as three different types of relations. A homogeneous network cannot always fully incorporate multiple relationships in a multi-graph as shown in [73]. Therefore, to effectively model these flow multi-graphs, we have designed a heterogeneous GNN network capable of handling each of these three relations and the different types of nodes in the graph. This model is an agglomeration of three different GNNs to model each flow graph (data flow, control flow, and call flow). Each of these three sub-networks are homogeneous in nature as they are expected to model only a single relation and the associated nodes. Our heterogeneous GNN, consisting of three layers, models these flow graph representations and their node features as shown in Figure 3. Each homogeneous sub-network in the heterogeneous GNN network in this paper is a Gated Graph Convolutional Network [48] with a "mean" aggregation scheme to group the node embeddings from each relation. This heterogeneous GNN model consists of approximately 350K trainable paramters.

***Modeling code vectors using Denoising Autoencoders.*** The dataset of code vectors obtained in Section 3.1 takes tabular structure, where each row in the table represents a sequence of code vectors. The usual practice while working with tabular data is to use gradient boosted techniques such as XGBoost [18], LightGBM [42]. Indeed such an approach has been used in [81] for their modeling tasks. However, due to the inherent difficulty of adapting GNNs and XGBoost as part of the same infrastructure, we have used denoising autoencoders as an alternative. The best submission on a Kaggle competition with a tabular dataset [38] highlighted DAEs as a possible alternative to gradient boosted algorithms.

This is a self-supervised technique, where we initially collect the code vectors in the training set and pass it through the DAE model. Our simple DAE model consists of five fully connected ANN/MLP layers, where the first two layers are the encoder layers, followed by a code layer, followed by two decoder layers. As mentioned in Section 2.4, the dimensions of the input and output layers are identical. Prior to modeling, the input data is scaled into a standard normal distribution using Gaussian rank scaling. During the encoding phase of the encoder-decoder architecture, we introduce "swap noise" into the dataset. Imagine a table of data, where for any given column, a value in that column is replaced by a randomly sampled value from the same column, such that 10% of values in a column has been modified. This modified data is then input to the encoding phase of the DAE with the target of predicting the correct input. This technique allows the DAE model, with approximately 1.8M trainable parameters, to better learn the distribution of the dataset.

***Fully Connected Tuning.*** As discussed in Section 2.5, late fusion

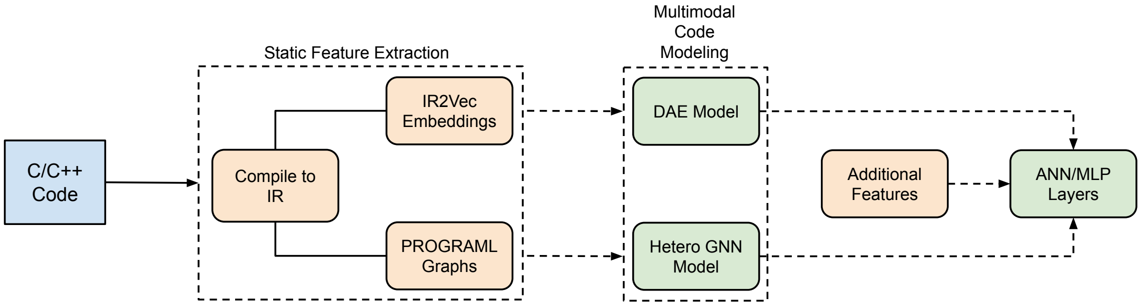


**Figure 3: MGA Pipeline: An overview of the tasks in our Heterogeneous GNN based Multimodal DL tuner. The compiled IR is passed through IR2Vec and PROGRAML. The outputs are then passed through the DAE and heterogeneous GNN models respectively. Additional features are experiment specific as shown in Sections 4.1 and 4.2.**

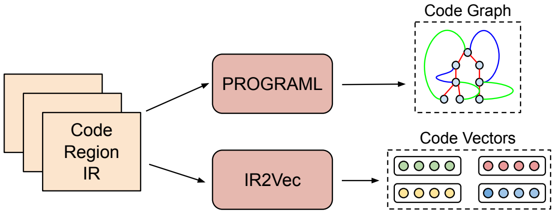


**Figure 4: Multimodal code representation**

**Table 1: Deep Learning Model Architecture.**

| Sub-Model | Layers | Network | Activation |
|---|---|---|---|
| HeteroConv | 3 | Gated Graph Convolution | ReLU |
| DAE. | 5 | Fully Connected (Encoder-Decoder) | ReLU, Sigmoid |
| Classifier | 3 | Fully Connected | ReLU |

techniques were used to fuse the outputs from each modeled modality. The output tensors from the last level layers of the GNN and DAE networks are initially concatenated. This fused tensor is then concatenated with additional experiment-specific features as mentioned before and detailed in Section 4. These additional features are performance counters for the OpenMP experiments in Section 4.1. For the OpenCL experiments, these are the transfer size and workgroup size. Prior to concatenation, these features are normalized and scaled to a [0,1] range. This feature vector is then fed as input to the fully connected (dense/MLP) layers [66] as can be seen in Figure 3. These layers are trained with the target of identifying the best runtime configurations. The fully connected MLP layers model all the aggregated features and classifies the loops/kernels and corresponding inputs to the appropriate configurations. Our fully connected network consists of only one hidden layer and around 200 trainable parameters. We have consciously designed a small network to reduce training time at source. We show later that such a multimodal modeling technique produces much better results than other auto-tuners and state-of-the-art code representations using a single modality.

## 4 EXPERIMENTS

We validate our hypothesis on tasks using two programming models, OpenMP and OpenCL. We chose to work primarily with OpenMP as it is widely used in the parallel programming community, and can be easily compiled to their intermediate representations (IRs). We additionally used OpenCL to check the strength of our code representation and modeling technique. We have worked with multiple benchmarks and various input sizes to closely mimic real world scenarios. We have compared our experiments with the state-of-the-art tools available in literature. The setups for each experiment are detailed in the corresponding sections.

***Experimental Systems and Software.*** The experiments in Section 4.1.3 targets an 8-core Intel i7-10700K (Comet Lake) processor. The experiments in Section 4.1.4 target a 10 core Intel Xeon Silver 4114 (Skylake) processor with two hyper-threads per core. We work with a dataset generated on Intel Core i7-3820 CPU and AMD Tahiti 7970 and NVIDIA GTX 970 GPUs in Section 4.2. Code regions are compiled and extracted using Clang tools. PyTorch and Pytorch Geometric libraries were used for building our DL models.

***Identifying and Selecting Benchmarks.*** The first step in our pipeline centers around the appropriate selection of benchmarks for experimentation. The benchmark applications were selected to have sufficient variability amongst them. We have used loops and kernels from multiple applications targeting domains ranging from arithmetic solvers to those targeting linear algebra, data mining, bioinformatics, fluid dynamics, image processing and others. For the OpenMP experiments, we used kernels from STREAM [54], DataRaceBench [49], Polybench [60], NAS [12], Rodinia [16, 17], LULESH [39, 40], XSBench [78], RSBench [77], miniFE [35], miniAMR [65], and Quicksilver [46]. The OpenCL experiments use kernels from the AMD SDK [7], NPB [67], NVIDIA SDK [19], Parboil [71], Polybench [30], Rodinia [16] and SHOC [24] benchmark suites. The benchmark applications used across all experiments are listed in Table 2.

### 4.1 OpenMP Tuning

In this section we have tried tuning OpenMP runtime parameters for OpenMP loops. These parameters can highly impact performance on CPUs and we try to identify those configurations that lead to the fastest executions. We initially compile the code to their IRs

Table 2: List of benchmarks used in experiments

| Benchmark Suite | Applications Selected |
|---|---|
| Polybench [87] | 2mm, 3mm, atax, adi, bicg, cholesky, convolution-2d, convolution-3d, correlation, covariance, doitgen, durbin, fdtd-2d, fdtd-apml, gemm, gemver, gesummv, gramschmidt, jacobi-1d, jacobi-2d, lu, mvt, seidel-2d, symm, syrk, syr2k, trisolv, trmm |
| Rodinia [16, 17] | b+tree, backprop, bfs, cfd, gaussian, hotspot, kmeans, lavaMD, leukocyte, lud, nn, nw, needle, particlefilter, pathfinder, srad, streamcluster |
| NAS [12] | BT, CG, EP, FT, LU, MG, SP |
| STREAM [52, 53] | stream.c |
| DataRaceBench [49] | DRB045, DRB046, DRB061, DRB062, DRB093, DRB094, DRB121 |
| AMD SDK [7] | BinomialOption, BitonicSort, BlackScholes, FastWalshTransform, FloydWarshall, MatrixMultiplication, Matrix-Transpose, PrefixSum, Reduction, ScanLargeArrays, SimpleConvolution, SobelFilter |
| NVIDIA SDK [19] | DotProduct, FDTD3D, MatVecMul, MatrixMul, MersenneTwister, VectorAdd |
| Parboil [71] | BFS, cutcp, lbm, sad, spmv, stencil |
| SHOC [24] | BFS, FFT, GEMM, MD, MD5, Reduction, S3D, Scan, Sort, Spmv, Stencil2D, Triad |
| Proxy/Mini Applications | LULESH [39, 40], XSBench [78], RSBench [77], miniFE [35], miniAMR [65], Quicksilver [46] |

and model them as described in Section 3. We augment these static code features with dynamic features in the form of performance counters. Performance counters are necessary for this experiment to help analyze the impact of various inputs on an OpenMP loop.

*4.1.1 Data Collection and Preprocessing.* Initially, each application is instrumented to accept variable input at runtime. Additionally each OpenMP loop is instrumented to call appropriate PAPI [57] APIs for profiling purposes. Each instrumented application is then profiled for each input size and configuration. This is a one time cost of creating the dataset and identifying the best configurations as labels of the dataset. A major bottleneck of this process is the large number of available performance counters. All systems used for this experiment reports >50 preset counters. We collected 20 PAPI counters based on the ideas presented in [1–3] for the Polybench suite. We extend and update these techniques to build our own dataset of OpenMP loop signatures. For each loop, we used 30 input sizes ranging from 3.5KB to 0.5GB. Profiling with multiple inputs provides insight into how these inputs impact the execution behavior of each OpenMP loop. The input sizes were selected with the intention of stressing each of the three cache levels (L1, L2, L3) to different degrees. This type of input-driven profiling lets us explore how varying runtime parameters can help alleviate latency issues. However, including all counters while training our tuning model leads to a feature explosion and negatively impacts model convergence. To improve model convergence, we used Pearson's correlation [14] and identified five performance counters that are most correlated to execution time, and used these for training purposes. For the remaining applications, we only profile them to collect these five counters. For an application with multiple OpenMP loops, the associated counters and execution times are collected in a single run. This implicitly accounts for the effect on hardware components a preceding loop might have on succeeding ones. These steps reduce the profiling cost to a large degree. The selected performance counters are L1, L2 cache misses, L3 load misses, number of retired branch instructions, and mispredicted branches across all loops, inputs and experiments. Overall, we collected more than 150k samples for the OpenMP experiments (Section 4.1).

*4.1.2 Setting up Baselines.* In this section, we have compared our results with three autotuners, ytopt, OpenTuner, and BLISS and two state-of-the-art code representations PROGRAML and IR2Vec. ytopt [9] and BLISS [64] are autotuners based on Bayesian optimization. OpenTuner [8] is a search-based autotuner which employs various search techniques such as AUC Bandit, Nelder-Mead, Torczon hillclimbers, etc. ytopt and OpenTuner have been previously used for a variety of tuning tasks [32, 36, 44, 85, 88] and were hence chosen as baselines for this paper. BLISS represents a more recent state-of-the-art autotuner based on Bayesian optimization. We have also compared against unimodal DL approaches that uses only PROGRAML [20] or IR2Vec [81] as the code representations of choice. In addition, we have also compared our results with brute-force tuning results (*Oracle*), where every possible configuration in the search space was evaluated to identify the absolute best configuration. For comparisons with ytopt, OpenTuner, and BLISS, the search and tuning methods specified in these tools were followed without any changes. For each tool, the search space and *compilation* and *run* commands were specified as per requirement. For ytopt and BLISS, the number of maximum evaluations was set to ten, and for OpenTuner, an upper bound time limit of 180 seconds was set.

*4.1.3 OpenMP Thread Prediction.* One of the most widely used techniques for improving the performance of OpenMP code is by varying the thread-level parallelism. Simply allocating more threads to a workload might not always produce the best results as shown in Figure 1.

The heterogeneous GNN model used across our experiments consists of three homogeneous GNN models. We experimented with a few popular graph neural networks: graph convolution networks (GCNs) [43], graph attention networks [80], GraphSAGE [34], and gated graph neural networks (GGNN) [48]. We observed that using GGNNs for modeling each relation in the flow graphs produces the

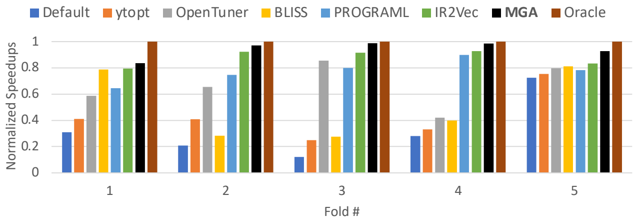


**Figure 5: Thread Prediction: Normalized speedups (with respect to oracle speedups) per validation fold. The MGA tuner produces speedups of 2.71×, 4.68×, 8.09×, 3.51×, 1.31× for each fold over default execution with all threads. Default speedup is always 1.0×.[Higher is better]**

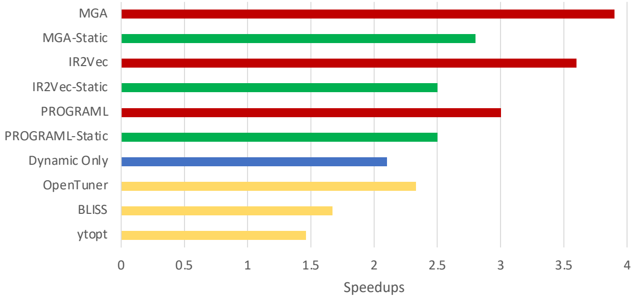


**Figure 6: Thread Prediction: Impact of static and dynamic features. Red bars use both static and dynamic features. Green bars use only static features. The blue bar uses only dynamic features (perf. counters). Yellow bars are existing tuners in literature (added for comparison) [Higher is better].**

best end results. The IR2Vec embeddings are modeled using DAE layers with Sigmoid activation function as described in Section 3.2. The output tensors from these models are then fused and concatenated with the performance counters and fed into the MLP layers to predict the number of threads. This model is optimized with the AdamW optimizer [50].

To evaluate our model performance, we perform 5-fold cross validation. Here, we run the same experiment five times, where the five validation folds are mutually exclusive sets and the union of these five sets equal the set of all loops in the dataset. For each validation fold, the other loops in the dataset (four-fifth of all loops) are assigned to the training set. The model is then iteratively trained and validated five times to ensure coverage of all loops in the dataset. In the absence of a designated, representative test set, k-fold cross validation allows us to test the skill of the model on unseen data. Our model achieves geometric mean accuracy of 86% in identifying the best threads across five folds.

The results in Figure 5 show that our approach performs better compared to other approaches. For each loop in the validation set, we use the predicted configuration for each input to obtain the execution time for that combination, and calculate the speedup (speedup = $\frac{Runtime_{default}}{Runtime_{new}}$) with respect to the default execution time. We repeat this process for each loop in the validation set. The geometric mean of these speedups is presented as each bar for each fold in Figure 5.

***Analyzing design choices and results.*** As mentioned before, we compared our approach with two unimodal approaches using PROGRAML and IR2Vec as the code representation of choice along with the tuners ytopt, OpenTuner, and BLISS. The tuners using unimodal code representation were modeled using the same underlying sub-model as used in MGA. The unimodal approach based only on PROGRAML graph features used the same GNN architecture used in our work. Similarly, the unimodal approach built on IR2Vec features used the DAE architecture designed as part the MGA tuner. The hyper-parameters for each of these unimodal DL approaches were tuned to the best of our ability to maximize performance. Such feature extraction and modeling techniques were employed to approximately quantify the advantages of our multimodal modeling approach in comparison to the other unimodal DL approaches and also serves implicitly as an *ablation study* to show the benefits of multimodality compared to individual state-of-the-art code representations. We treated ytopt, OpenTuner, and BLISS as black boxes and simply provided the target metric and other necessary information as mentioned in Section 4.1.2. In this section, speedups are calculated with respect to the execution time with default OpenMP configurations (all threads, static scheduling, compiler calculated chunk size). In three out of five folds, our approach produced normalized speedups (with respect to Oracle speedups) of $\geq 0.95\times$, and in one out of five folds normalized speedup between 0.9× and 0.95× of the oracle speedups. The IR2Vec tuner led to normalized speedups of $\geq 0.9\times$, but $< 0.95\times$ in three out of five folds, and had normalized speedups $< 0.85\times$ in the remaining folds. The PROGRAML tuner produced normalized speedups of $> 0.85\times$ in one out of five folds. As seen in Figure 5, ytopt produced normalized speedups $> 0.75\times$ in one out of five folds. OpenTuner and BLISS produced speedups of $> 0.75\times$ in two out of five folds. Our approach only shows reduced performance gains in one fold. This is primarily due to the presence of the trisolv kernel from Polybench. The serial version of trisolv has better performance than the parallel version used in this paper. This worsens the result of fold one, as the DL model does not see similar trends for other loops based on code modeling and execution behavior.

Amongst the considered tuning approaches, our method came closest to the Oracle predictions. ytopt, OpenTuner, BLISS, the PROGRAML tuner, the IR2Vec tuner, and the MGA tuner produced geometric mean speedups of 1.46×, 2.33×, 1.67×, 2.79×, 3.17×, and 3.4× across all folds compared to oracle speedups of 3.62×. We believe that our approach is able to better capture and model code semantics and structure which aids the process of identifying "good" configurations in comparison to the other unimodal DL approaches. The difference in performance between the MGA tuner, and PROGRAML and IR2Vec tuners point to this.

***Importance of dynamic information.*** Performance profiling is an overhead of our approach. However, we posit that modeling performance counters is essential for such DL-based tuning. As the code features are *static* in nature, these cannot capture/convey to the model runtime/execution information. Rather than handcrafting such input related features (often requires expert knowledge), performance counters were used to capture the impact of inputs in

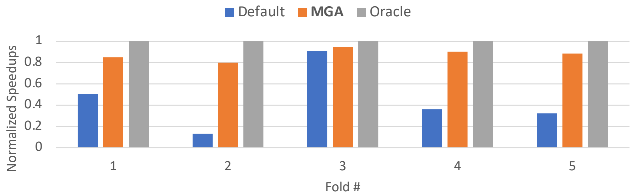


**Figure 7: Thread Prediction on *unseen loops and input size.* Speedups normalized with respect to Oracle. The MGA tuner produces 1.68×, 6.0×, 1.04×, 2.5×, 2.73× speedups across five folds over default execution. [Higher is better]**

an automated fashion. We performed a set of studies to validate this claim. We trained three DL models with only static features and observed performance degradation when performance counters were not a part of the feature set. The results from the validation set obtained by performing a randomized 80/20 split are shown in Figure 6. Compared to achieved speedups of 3.9×, 3.6×, and 3.0× by the MGA, IR2Vec and PROGRAML models (uses both static and dynamic features) respectively, the speedups fell to 2.8×, 2.5×, and 2.5× without performance counters. This is expected as these static features do not explicitly provide information about the impact of varied inputs on execution. In addition, we also train a model with only dynamic features. It showed the smallest speedups amongst all the DL-tuners designed in this paper, achieving speedups of only 2.1×. Therefore, using static *and* dynamic features together lead to the best results.

***Varying Input Sizes.*** To evaluate the generalizability of our method, this section primarily evaluates how our model performs when *both loops and input sizes are unknown.* We initially selected at random 20% of the 30 input sizes considered in this paper, and set it aside for validation. We then split the loops using 5-fold validation as described before. Following this process, each validation fold now consists of unseen loops *and* the unknown input sizes set aside before. However, to reduce bias and preserve generalizability, the loops in these validation folds are different from the validation folds in the previous experiment (e.g. validation loops in fold one of this experiment is different from the validation loops in fold one of previous experiments). In the previous experiments, only the OpenMP loops were unknown in the validation set. The model had been trained on the training set of OpenMP loops and all input sizes. In contrast, in this experiment, the model is trained on the OpenMP loops in the training set and 80% of the input sizes. The loops in the validation sets and the unknown inputs are tested in this experiment and the results are shown in Figure 7. We observe that our model performs well producing geometric mean speedups of 2.35× across all folds, compared to mean oracle speedups of 2.68×. There is some performance drop as input sizes highly impact performance counters and the best runtime configurations. Without prior knowledge of program behavior at these input sizes, the model performance suffers.

### 4.1.4 *Scaling up to a Larger Search Space.*

The experiments performed in previous sections have achieved good results. However, those search spaces are fairly small. In order to assess the scalability of our approach to larger search spaces, we experimented with tuning the number of threads, scheduling policy, and chunk sizes at the same time. The search space is defined in Table 3 using ideas from [10, 11]. In this experiment, we have used a smaller subset of the benchmarks considered before and worked on loops from Polybench and Rodinia. We have additionally experimented with LULESH [39, 40], XSBench [78], RSBench [77], miniFE [35], miniAMR [65], and Quicksilver [46] applications. For this experiment, we performed *leave-one-out validation* instead of 5-fold validation, to better show the results of each application on a larger search space. In this method of validation, we leave out data associated with one benchmark application (all loops in this application are present in the validation set) as the validation set and train our model on the rest. This process is repeated for each considered application. As shown in Figure 8, this leads to normalized speedups of ≥ 0.95× of the oracle speedups in 26 out of 35 applications, and ≥ 0.85× normalized speedups in 33 out of 35 applications. trisolv is the worst performing application due to reasons discussed in Section 4.1.3. Our approach outperforms ytopt, OpenTuner, and BLISS in 32, 33, and 30 cases out of 35. ytopt, OpenTuner, and BLISS produce > 0.95× of the oracle speedups in 11, 4, and 16 cases out of 35. Overall, our model produces geometric mean speedups of around 2.01× compared to oracle speedups of 2.13×. The improvement in performance of most kernels can be attributed to better cache performance, branch predictions, and load balancing. We show the impact of using the predicted OpenMP configurations for this system on cache misses, clock cycles, and branch mispredictions compared to the default configuration of using all threads and static scheduling in Figure 9 for the 2mm kernel. There is a clear relation between improved performance and reduced cache misses, and branch mispredictions. In most kernels used in this paper, the profitable configurations lead to improvements in most of these factors, leading to improved performance.

**Table 3: Search Space for Experiment in Section 4.1.4**

| OpenMP **Parameter** | **Parameter Values** |
|---|---|
| Threads | 1, 2, 4, 8, 12, 16, 20 |
| Scheduling Policies | *static*, *dynamic*, *guided* |
| Chunk Sizes | 1, 8, 32, 64, 128, 256, 512 |

### 4.1.5 *Analyzing μ-architecture Portability.*

In this section, we analyzed if our auto-tuner can predict the number of threads on other μ-architectures. We re-used the model in Section 4.1.3 (trained on data from Comet Lake μ-architecture) to predict the number of threads on single-socket 8 core systems belonging to the Broadwell and SandyBridge μ-architecture (access provided by Cloudlabs infrastructure [25]). Limiting the scope of this experiment to testing hardware portability to single-socket 8 core systems helps us to directly use a pre-trained model without additional training (different core/socket count would necessitate re-training). Static code graphs, sequential code vectors, and the performance counters from the target systems (Broadwell/Sand Bridge) were passed as inputs to the pre-trained model. We validated this approach on 25 kernels from the PolyBench benchmark with STANDARD and LARGE inputs. Similar to Section 4.1.4, we perform *leave-one-out validation* for this experiment. The data from the Comet Lake

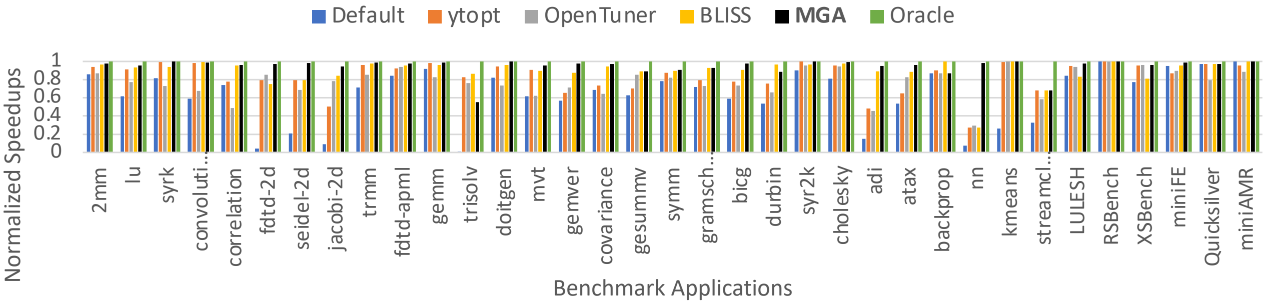


**Figure 8: Normalized speedups (w.r.t. oracle) for each application for Section 4.1.4 experiments. [Higher is better]**

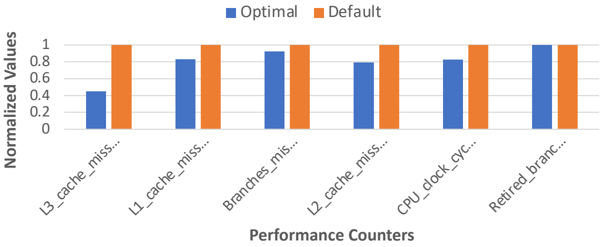


**Figure 9: Normalized performance counter values for `2mm` benchmark: default (20 threads, static scheduling, dynamically calculated chunk sizes) vs predicted configuration (16 threads, dynamic scheduling, chunks of 8). [Lower is better]**

system was always used for training in this experiment. For each validation fold, the validation kernel was executed twice on the target $\mu$-architecture to collect the necessary counters. The L1, L2, and L3 cache counters were then computed as a function of the system cache sizes relative to the system on which the training data is collected (e.g. L1 cache misses for Sandy Bridge are computed as $\frac{L1\_CM \times L1\_cache\_size_{SandyBridge}}{L1\_cache\_size_{CometLake}}$). The branch misprediction counters were divided by the number of reference clock cycles. These counters were then normalized to a [0,1] scale and fed into the model to predict the number of the threads. This process completely removes the overhead of re-training models for other similar $\mu$-architectures.

As shown in Figure 10, our approach performs well while predicting the number of threads on Broadwell and Sandy Bridge with a model trained on data from Comet Lake. In most cases, the predicted configurations lead to optimal/better performance. We observed that only using static features leads to the model making similar predictions for Comet Lake, Broadwell, and Sandy Bridge. Without modeling performance counters, the predictions for unseen loops are what it would be for the training $\mu$-architecture. This led to degraded performance on the target $\mu$-architectures.

***Observations and Analysis.*** For these experiments, our approach produces the best results overall. Search-based autotuners have been commonly used for tuning tasks in the HPC community. However, these do need to execute code multiple times to identify profitable configurations from the search space. Recently, Bayesian optimization (BO) based tuners such as `ytopt` and `BLISS` have built and improved on BO techniques to further improve the tuning process. As an active learning technique, these need less executions and evaluations by making smart data selection choices. Indeed as shown in [64], such BO-based tuners reach closest to Oracle predictions in comparison to search-based tools such as `OpenTuner` with lesser number of evaluations. However, BO-based tuners also require multiple code executions to effectively train their surrogate models, and are usually application and input specific, i.e. in general, these tuners need to be trained for each combination of code and input. In contrast, our trained model is neither application or input size specific. It is trained on data from multiple loops and inputs. During inference, our model is expected to predict “good" configurations for previously *unknown* applications and/or inputs. During inference, an application should be compiled to its IR and run only twice, irrespective of the size of the search space, to gather the performance counters which are then fed into the `MGA` model as inputs. This overhead is much less than existing tuners considered in this paper, but still produces better results. The number of maximum evaluations defined for `ytopt`, `OpenTuner`, and `BLISS` were limited to a small number as mentioned in Section 4.1.2. This was done to perform a fair comparison with our tool. Increasing the number of evaluations/executions for these tuners do lead to improved results at a higher tuning cost.

As all deep learning tasks, our approach also suffers a training overhead. We offset this overhead to some degree by designing simple models with only a few layers. These efforts led to training time of around 10 minutes per epoch with a `batch_size` of 32 on an 8-core Intel CometLake CPU for ~ 140K samples. During inference, individual predictions take around 200$\mu$s on the same CPU.

## 4.2 OpenCL Tuning

`OpenCL` is another programming model that is widely used in parallel programming. With this experiment we aim to validate if our approach works for a compiler optimization task for another IR-based programming model. Heterogeneous device mapping is a commonly used task to validate the effectiveness of code representations and modeling. Grewe et al. [31] proposed the device mapping task to map `OpenCL` kernels to the CPU or GPU. This task has been used in later works [13, 20, 81] to evaluate their performance. We also use this task to evaluate the effectiveness of our approach.

*4.2.1 Dataset.* We use the dataset published by Ben-Nun et al. [13] for this experiment. It has 256 unique `OpenCL` kernels from seven

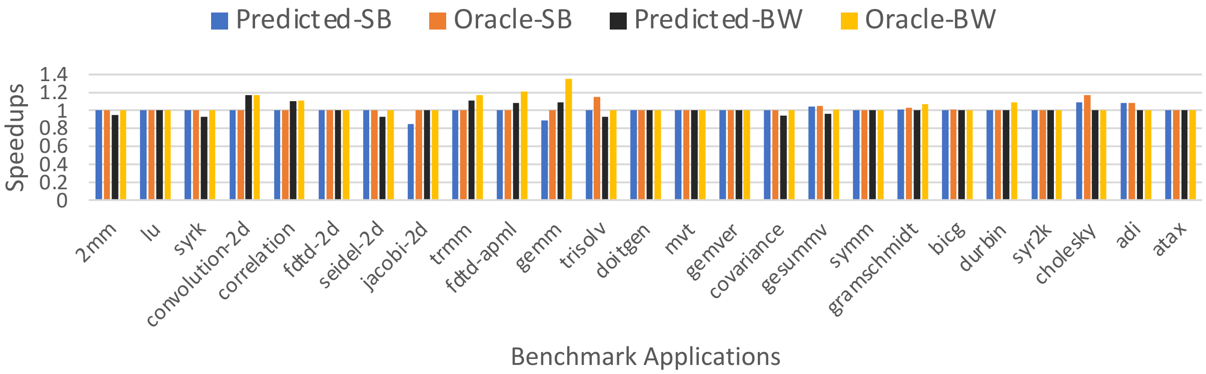


**Figure 10: Speedups for experiment in Section 4.1.5. Predictions for Broadwell (BW) and SandyBridge (SB) by model trained on data from Comet Lake [Higher is better]**

benchmark suites comprising of AMD SDK, NPB, NVIDIA SDK, Parboil, Polybench, Rodinia and SHOC. The data size and workgroup size were varied for each kernel to obtain a labeled dataset with 670 CPU- or GPU-labeled data points for each of the two devices, AMD Tahiti 7970 and NVIDIA 970. As this is a published dataset, no modifications were made to it, and performance counters have *not* been used in this experiment.

*4.2.2 Heterogeneous Device Mapping.* For modeling purposes, we use similar techniques used in the previous section. We initially use the extracted IR of the `OpenCL` kernels in the dataset. The IRs are then passed through `PROGRAML` and `IR2Vec` to obtain the code graphs and code vectors. As before, the code graphs are modeled using Heterogeneous GNNs and the code vectors are modeled using DAEs. For this experiment, the modeled outputs from the GNN and DAE models are concatenated as before. In addition, we also add transfer and workgroup sizes from the dataset to the feature set before passing the feature set onto the fully connected MLP layers. Following the techniques used in [20, 81], we have also used tenfold stratified cross-validation to evaluate our results. We were able to replicate the experiments reported in [81], and these results are used to compare our results. The results from [20] are used directly to compare our results. In this task, we validate if our approach can outperform the state-of-the-art to reinforce our hypothesis that existing code representations are good enough to be used in conjunction for better performance.

**Table 4: Accuracy: Heterogeneous device mapping (CPU/GPU). All numbers are in percentage. Numbers in parenthesis are percentage improvements in accuracy of `MGA` model over corresponding approaches.**

| **State-of-the-art** | **NVIDIA GPU** | **AMD GPU** |
|---|---|---|
| Grewe et al. [31] | 74.56 (31.3) | 70.29 (39.0) |
| DeepTune [21] | 80.88 (21.04) | 83.24 (17.37) |
| inst2Vec [13] | 82.65 (18.45) | 82.35 (18.64) |
| PROGRAML [20] | 80 (22.38) | 86.6 (12.82) |
| IR2Vec [81] | 89.68 (9.17) | 92.82 (5.26) |
| **MGA** (ours) | **97.9** | **97.7** |

Our experimental setup leads to state-of-the-art results in identifying the correct device. We achieve accuracy of 97.9% and F1-score of 0.98 in identifying the best device on the NVIDIA GPU. On the AMD GPU, we achieve accuracy and F1-score of 97.7% and 0.97. In comparison, PROGRAML achieves accuracies of 80% and 86.6% on the NVIDIA and AMD GPUs and corresponding F1-scores of 0.88 and 0.8. IR2Vec (flow-aware representation) achieves accuracies of 89.68% and 92.82% on the NVIDIA and AMD GPUs. Comparisons with other works on this dataset are shown in Table 4. The accuracy numbers for Grewe at al. [31], DeepTune [21], and inst2vec [13] are cited from [81].

We have also analyzed performance improvements due to the predictions by our model. The speedups are calculated in comparison to static mappings as done in [81]. On the NVIDIA 970 system, our approach leads to speedups of 1.3× compared to oracle speedups of 1.34×. The oracle speedups are calculated by analyzing the execution time on the best device and comparing it to the static mapping baseline. In comparison, the predictions in [81] led to speedups of 1.26×. On the AMD Tahiti system, our predictions lead to speedups of 1.62× compared to speedups of 1.58× produced by IR2Vec [81] and oracle speedups of 1.66×.

***Observations and Analysis.*** We have shown in this section that our approach produces better results than the state-of-the-art in this field without the need of a completely new code representation technique. We analyzed our model's predictions to identify those cases where our model outperformed the state-of-the-art. We were only able to replicate the experiments in [81] (best results in existing literature) and our observations are with respect to this paper. Our overall performance was better as our edge case predictions were better. We noticed that our model outperformed in corner cases where kernels with small inputs were mapped to the GPU, and kernels with large inputs were mapped to the CPU. As an illustrative example, consider the *makea* kernel from the `CG` benchmark in NPB. In the dataset, this kernel gets mapped to a GPU with a small input class S, whereas the same kernel with much larger input class C gets mapped to the CPU. This behavior can be due to the presence of multiple function calls from inside a loop. The called functions, also have parallel loops in them. This, we believe, creates an overhead which leads to faster execution on the CPU for larger inputs. For the smaller inputs, the number of function calls are much less which does not create a bottleneck for GPU execution, leading to much faster execution on GPUs. The MGA model is able to identify such edge cases as our approach can capture the characteristics of individual instructions and arguments along with the data, control, and call flows in a kernel.

## 5 RELATED WORK

This paper proposes a new multimodal code representation technique built on top of state-of-the-art representation techniques and its usage for DL based tuning of runtime configurations for `OpenMP` and `OpenCL` kernels. These programming models expose a number of configurations for runtime optimization. Thus auto-tuning is essential for identifying the optimum configurations.

There already exists a large body of research on tuning runtime/code parameters or configurations for parallel code [28, 37, 41, 58, 68, 74]. `OpenTuner` [8] and `ActiveHarmony` [74] are auto-tuning frameworks for domain-specific tuning that is much faster than exhaustive search-based auto-tuners. These tuners employ a variety of search techniques for search space exploration and optimizations.

An alternative to search-based auto-tuning is to use machine learning based approaches. Search-based auto-tuners mostly depend on manually or pre-defined heuristics to identify optimum points in the search space. Such an approach iteratively explores the search space to identify patterns that might point to profitable configurations in the search space. Such tuners, however, need to execute applications a number of times, which is often expensive. ML tuners can reduce this exploration because of its pre-training and ability to associate similarities between applications. To this end, [62, 83] propose machine learning based approaches to `OpenMP` autotuning. Artemis [84] is an automatic parameter tuning framework that uses machine learning to predict the execution parameters of parallel regions. `ytopt` [9] is an evolution of the work in [68] which iterates over a set of user-defined configurations and their possible values to arrive at a tuned configuration. These approaches are often domain or application specific. Although often faster than search-based alternatives, these do need multiple code executions as evidenced by our experiments with `ytopt` [9].

Deep learning provides another alternative to the aforementioned techniques. A suitable code representation technique is essential for such deep learning based code modeling. To this end, several code representations have been proposed [5, 6, 13, 15, 20, 22, 23, 63, 81], which have been used to good effect for several optimization tasks such as heterogeneous device mapping, thread coarsening factor, etc. to name a few. PROGRAML [20] and IR2Vec [81] are two state-of-the-art such code representations, which have addressed the shortcomings of seminal works in code representation such as `inst2vec` [13]. However, as mentioned before, each of these representations suffer from some limitations. Given the complexity of developing new code representation techniques, building on top of existing ones seems wise. Unlike the papers mentioned before, this study considers two code representations as two separate modalities for improving performance over unimodal approaches.

We have modeled our modalities using heterogeneous GNNs and DAEs. A few works such as [20, 26, 27, 75], in the recent past have successfully used GNNs for code modeling tasks. However, to the best of our knowledge, this is the first work that employs heterogeneous GNNs for such tasks. Additionally, we believe no other work has previously used denoising autoencoders to model code vectors and adapted multimodal learning for code representation learning.

## 6 DISCUSSION

Through this work, we have presented the idea of using heterogeneous GNNs, denoising autoencoders, and multimodal deep learning for the purpose of DL-based code modeling. We believe the next phase of innovation in performance optimization will come from deep learning approaches. As evidenced by works such as [13, 20, 21, 75, 81], deep learning has been successfully used for compiler and performance optimizations. Our experimental results in this paper further reinforce that belief. However, DL is not a "silver bullet" for all problems. These approaches do come with overheads in model training. To address this, we have consciously designed shallow networks to speed up training and inference at source. The strength of any deep learning model lies in the set of features it models. For the tasks considered in this paper, along with the static code features, it was essential to incorporate runtime/dynamic features into the feature set. A good option is to carefully handcraft such features for each experiment. But this requires expert intervention and is costly. Thus a limited number performance counters were incorporated to represent such dynamic features in a more automated way. This allows us to easily collect and represent some runtime features while keeping the profiling overhead constrained.

## 7 CONCLUSION AND FUTURE WORKS

The presented technique of utilizing varied code representations as different modalities is unique and promising for optimization tasks. The multimodal code representation outperforms both unimodal code representations considered in this paper. Our multimodal learner also performs well when faced with unknown code and inputs. This technique has led to us setting state-of-the-art results in the task of `OpenCL` device mapping, and to the development of a multimodal `OpenMP` tuner, producing better results than existing auto-tuners. We aim to incorporate transfer and reinforcement learning in future efforts for developing an online tuner with customizable search spaces and expand our work to GPUs and FPGAs.

## ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under Grant number 2211982. We would also like to thank the ResearchIT team [1] at Iowa State University for their constant support. This work was also supported by the Ministerio de Ciencia e Innovación MCIN AEI/10.13039/501100011033 under contract PID2020-113614RB-C21 and by the Catalan government under contract 2021 SGR 00574.

## REFERENCES

[1] Jordi Alcaraz, Anna Sikora, and Eduardo César. 2019. Hardware counters' space reduction for code region characterization. In *European Conference on Parallel Processing*. Springer, 74–86.

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

[2] Jordi Alcaraz, Steven Sleder, Ali TehraniJamsaz, Anna Sikora, Ali Jannesari, Joan Sorribes, and Eduardo Cesar. 2021. Building representative and balanced datasets of OpenMP parallel regions. In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 67–74.

[3] Jordi Alcaraz, Ali TehraniJamsaz, Akash Dutta, Anna Sikora, Ali Jannesari, Joan Sorribes, and Eduardo Cesar. 2022. Predicting number of threads using balanced datasets for openMP regions. *Computing* (2022), 1–19.

[4] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[5] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices* 53, 4 (2018), 404–419.

[7] AMD. [n.d.]. AMD OpenCL accelerated parallel processing SDK. https://developer.amd.com/amd-accelerated-parallel-processing-app-sdk/.

[8] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.

[9] P Balaprakash, R Egele, and P Hovland. 2020. ytopt. https://github.com/ytopt-team/ytopt (GitHub repository). Argonne National Laboratory. (2020).

[10] Md Abdullah Shahneous Bari, Nicholas Chaimov, Abid M Malik, Kevin A Huck, Barbara Chapman, Allen D Malony, and Osman Sarood. 2016. Arcs: Adaptive runtime configuration selection for power-constrained openmp applications. In *2016 IEEE international conference on cluster computing (CLUSTER)*. IEEE.

[11] Md Abdullah Shahneous Bari, Abid M Malik, Ahmad Qawasmeh, and Barbara Chapman. 2019. Performance and energy impact of OpenMP runtime configurations on power constrained systems. *Sustainable Computing: Informatics and Systems* 23 (2019), 1–12.

[12] E Barszcz, J Barton, L Dagum, P Frederickson, T Lasinski, R Schreiber, V Venkatakrishnan, S Weeratunga, D Bailey, D Browning, et al. 1991. The nas parallel benchmarks. In *The International Journal of Supercomputer Applications*. Citeseer.

[13] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. *Advances in Neural Information Processing Systems* 31 (2018), 3585–3597.

[14] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. In *Noise reduction in speech processing*. Springer, 1–4.

[15] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*. 201–211.

[16] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[17] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*. IEEE, 1–11.

[18] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, et al. 2015. Xgboost: extreme gradient boosting. *R package version 0.4-2* 1, 4 (2015), 1–4.

[19] NVIDIA Corporation. [n.d.]. CUDA. http://developer.nvidia.com/object/cuda.html.

[20] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael FP O'Boyle, and Hugh Leather. 2021. PROGRAML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *International Conference on Machine Learning*. PMLR, 2244–2253.

[21] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 219–232.

[22] Christopher Edward Cummins. 2020. Deep learning for compilers. (2020).

[23] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. 2018. A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921* (2018).

[24] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*. 63–74.

[25] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[26] Akash Dutta, Jordi Alcaraz, Ali TehraniJamsaz, Anna Sikora, Eduardo Cesar, and Ali Jannesari. 2022. Pattern-based autotuning of openmp loops using graph neural networks. In *2022 IEEE/ACM International Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)*. IEEE, 26–31.

[27] Akash Dutta, Jee Choi, and Ali Jannesari. 2023. Power Constrained Autotuning using Graph Neural Networks. In *IPDPS 2023-37th IEEE International Parallel & Distributed Processing Symposium*.

[28] Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Cristina Silvano. 2018. Margot: a dynamic autotuning framework for self-aware approximate computing. *IEEE transactions on computers* 68, 5 (2018), 713–728.

[29] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 1263–1272.

[30] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 innovative parallel computing (InPar)*. Ieee, 1–10.

[31] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. 2013. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–10.

[32] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 100–112.

[33] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 242–255.

[34] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[35] Si Hammond, Christian Trott, and Noah Evans. 2022. miniFE. https://github.com/Mantevo/miniFE. *GitHub repository* (2022).

[36] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. 2020. Unicorn: Runtime provenance-based detector for advanced persistent threats. *arXiv preprint arXiv:2001.01525* (2020).

[37] Zia Ul Huda, Rohit Atre, Ali Jannesari, and Felix Wolf. 2016. Automatic parallel pattern detection in the algorithm structure design space. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 43–52.

[38] Michael Jahrer. 2017. Porto Seguro's Safe Driver Prediction. https://www.kaggle.com/c/porto-seguro-safe-driver-prediction/discussion/44629. (2017).

[39] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. 2013. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*. Boston, USA.

[40] Ian Karlin, Jeff Keasler, and J Robert Neely. 2013. *Lulesh 2.0 updates and changes*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[41] Jakub Katarzyński and Maciej Cytowski. 2014. Towards autotuning of OpenMP applications on multicore architectures. *arXiv preprint arXiv:1401.4063* (2014).

[42] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[43] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[44] Jaehoon Koo, Prasanna Balaprakash, Michael Kruse, Xingfu Wu, Paul Hovland, and Mary Hall. 2021. Customized Monte Carlo Tree Search for LLVM/Polly's Composable Loop Optimization Transformations. *arXiv preprint arXiv:2105.04555* (2021).

[45] Sameer Kulkarni, John Cavazos, Christian Wimmer, and Douglas Simon. 2013. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–12.

[46] Lawrence Livermore National Lab. 2022. Quicksilver. https://github.com/LLNL/Quicksilver.

[47] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*. PMLR, 3835–3845.

[48] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[49] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[50] Ilya Loshchilov and Frank Hutter. 2018. Fixing weight decay regularization in adam. (2018).

[51] Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 455–466.

[52] John D. McCalpin. 1991-2007. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Technical Report. University of Virginia, Charlottesville, Virginia. http://www.cs.virginia.edu/stream/ A continually updated technical report. http://www.cs.virginia.edu/stream/.

[53] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[54] John D McCalpin. 1995. Stream benchmark. *Link: www. cs. virginia. edu/stream/ref. html# what* 22, 7 (1995).

[55] Charith Mendis, Cambridge Yang, Yewen Pu, Dr Amarasinghe, Michael Carbin, et al. 2019. Compiler auto-vectorization with imitation learning. *Advances in Neural Information Processing Systems* 32 (2019).

[56] Harshitha Menon, Abhinav Bhatele, and Todd Gamblin. 2020. Auto-tuning parameter choices in HPC applications using Bayesian optimization. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 831–840.

[57] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710. Citeseer.

[58] Dheya Mustafa, Rudolf Eigenmann, et al. 2011. Performance analysis and tuning of automatically parallelized OpenMP applications. In *International Workshop on OpenMP*. Springer, 151–164.

[59] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. 2011. Multimodal deep learning. In *ICML*.

[60] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012), 1–1.

[61] Dhanesh Ramachandram and Graham W Taylor. 2017. Deep multimodal learning: A survey on recent advances and trends. *IEEE signal processing magazine* 34, 6 (2017), 96–108.

[62] Piyumi Rameshka, Pasindu Senanayake, Thulana Kannangara, Praveen Seneviratne, Sanath Jayasena, Tharindu Rusira, and Mary Hall. 2019. Rigel: A Framework for OpenMP PerformanceTuning. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2093–2102.

[63] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from" big code". *ACM SIGPLAN Notices* 50, 1 (2015), 111–124.

[64] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. 2021. Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1280–1295.

[65] Aparna Sasidharan and Marc Snir. 2016. MiniAMR-A miniapp for Adaptive Mesh Refinement. (2016).

[66] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.

[67] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *2011 IEEE international symposium on workload characterization (IISWC)*. IEEE, 137–148.

[68] Vinu Sreenivasan, Rajath Javali, Mary Hall, Prasanna Balaprakash, Thomas RW Scogland, and Bronis R de Supinski. 2019. A framework for enabling OpenMP autotuning. In *International Workshop on OpenMP*. Springer, 50–60.

[69] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. 2021. Value learning for throughput optimization of deep learning workloads. *Proceedings of Machine Learning and Systems* 3 (2021).

[70] Mark Stephenson and Saman Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *International symposium on code generation and optimization*. IEEE, 123–134.

[71] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.

[72] Jabeen Summaira, Xi Li, Amin Muhammad Shoib, Songyuan Li, and Jabbar Abdul. 2021. Recent Advances and Trends in Multimodal Deep Learning: A Review. *arXiv preprint arXiv:2105.11087* (2021).

[73] Jianing Sun and Yingxue Zhang. 2019. Multi-graph convolutional neural networks for representation learning in recommendation. In *IEEE ICDM*.

[74] Cristian Tapus, I-Hsin Chung, and Jeffrey K Hollingsworth. 2002. Active harmony: Towards automated performance tuning. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 44–44.

[75] Ali Tehranijamsaz, Mihail Popov, Akash Dutta, Emmanuelle Saillard, and Ali Jannesari. 2022. Learning Intermediate Representations using Graph Neural Networks for NUMA and Prefetchers Optimization. In *IPDPS 2022-36th IEEE International Parallel & Distributed Processing Symposium*.

[76] Jayaraman J Thiagarajan, Nikhil Jain, Rushil Anirudh, Alfredo Gimenez, Rahul Sridhar, Aniruddha Marathe, Tao Wang, Murali Emani, Abhinav Bhatele, and Todd Gamblin. 2018. Bootstrapping parameter space exploration for fast tuning. In *Proceedings of the 2018 international conference on supercomputing*. 385–395.

[77] John R Tramm, Andrew R Siegel, Benoit Forget, and Colin Josey. 2014. Performance analysis of a reduced data movement algorithm for neutron cross section data in monte carlo simulations. In *International Conference on Exascale Applications and Software*. Springer, 39–56.

[78] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)* (2014).

[79] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 207–216.

[80] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[81] S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. 2020. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 4 (2020), 1–27.

[82] Xiao Wang, Deyu Bo, Chuan Shi, Shaohua Fan, Yanfang Ye, and Philip S Yu. 2020. A survey on heterogeneous graph embedding: methods, techniques, applications and sources. *arXiv preprint arXiv:2011.14867* (2020).

[83] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael FP O'boyle. 2014. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 1 (2014), 1–26.

[84] Chad Wood, Giorgis Georgakoudis, David Beckingsale, David Poliakoff, Alfredo Gimenez, Kevin Huck, Allen Malony, and Todd Gamblin. 2021. Artemis: Automatic Runtime Tuning of Parallel Execution Parameters Using Machine Learning. In *International Conference on High Performance Computing*. Springer, 453–472.

[85] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Hal Finkel, Paul Hovland, Valerie Taylor, and Mary Hall. 2021. Autotuning PolyBench Benchmarks with LLVM Clang/Polly Loop Optimization Pragmas Using Bayesian Optimization (extended version). *arXiv preprint arXiv:2104.13242* (2021).

[86] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[87] Tomofumi Yuki and Louis-Noël Pouchet. 2015. Polybench 4.0.

[88] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.