MIREncoder: Multi-modal IR-based Pretrained Embeddings for Performance Optimizations

Akash Dutta
Iowa State University
United States of America
adutta@iastate.edu

Ali Jannesari
Iowa State University
United States of America
jannesar@iastate.edu

Abstract

One of the primary areas of interest in High Performance Computing is the improvement of performance of parallel workloads. Nowadays, compilable source code-based optimization tasks that employ deep learning often exploit LLVM Intermediate Representations (IRs) for extracting features from source code. Most such works target specific tasks, or are designed with a pre-defined set of heuristics. So far, pre-trained models are rare in this domain, but the possibilities have been widely discussed. Especially approaches mimicking large-language models (LLMs) have been proposed. But these have prohibitively large training costs. In this paper, we propose `MIREncoder`, a **M**ulti-modal **IR**-based Auto-**Encoder** that can be pre-trained to generate a learned embedding space to be used for downstream tasks by machine learning-based approaches. A multi-modal approach enables us to better extract features from compilable programs. It allows us to better model code syntax, semantics and structure. For code-based performance optimizations, these features are very important while making optimization decisions. A pre-trained model/embedding implicitly enables the usage of transfer learning, and helps move away from task-specific trained models. Additionally, a pre-trained model used for downstream performance optimization should itself have reduced overhead, and be easily usable. These considerations have led us to propose a modeling approach that i) understands code semantics and structure, ii) enables use of transfer learning, and iii) is small and simple enough to be easily re-purposed or reused even with low resource availability. Our evaluations will show that our proposed approach can outperform the state of the art while reducing overhead.

CCS Concepts

• **Computing methodologies** → **Parallel computing methodologies**; **Knowledge representation and reasoning**; **Supervised learning by classification**; **Neural networks**; **Modeling methodologies**.

Keywords

Pre-training, GNN, Multi-modal Modeling, Performance Optimization, Auto-tuning, LLVM

PACT '24, October 14–16, 2024, Long Beach, CA, USA

ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3676895

ACM Reference Format:
Akash Dutta and Ali Jannesari. 2024. MIREncoder: Multi-modal IR-based Pretrained Embeddings for Performance Optimizations. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3656019.3676895

1 Introduction

The complexity, scale, and heterogeneity of HPC hardware has increased significantly over the past several years improving performance over traditional multi-core systems. However, this has also opened up new opportunities of performance optimizations. Performance engineers and application developers devote considerable time in trying to tune and optimize hardware and software knobs. However, it is extremely difficult to adapt to a constantly changing landscape. Automated techniques are thus necessary to help optimize performance of HPC applications.

Prior Works. A large chunk of performance gains for parallel applications come from compiler optimizations, such as those seen in `LLVM` and `GCC`. Although such optimizations are painstakingly designed, it might not work in all cases due to the variety of applications seen in HPC. In addition to compiler-driven optimizations, runtime performance tuning by online auto-tuners [7, 42, 50, 58] also help identify configurations/parameters that might often be non-intuitive. Although this improves performance, it comes with significant tuning overhead.

Machine learning (ML) based techniques have also been widely used for such performance optimizations. Several works have used ML to model handcrafted features for specific tasks[1, 7, 13, 30, 43]. These handcrafted features are not universal and might not be suitable for other optimization tasks. To overcome these shortcomings, studies based on code representational learning were proposed. Most of these works proposed a means of representing source code in a way understandable by machine learning models. Various works[3–5] designed representations on top of source code for tasks such as variable misuse and method name prediction. However, such representations put a lot of emphasis on stylistic choices in source code, are language dependent, thus are not ideal candidates for performance optimization tasks of compilable source code. Our proposed approach can, on the other hand, work with multiple languages as shown later in Section 4.

These aforementioned representations are also not adept at capturing program dependencies. Thus LLVM IR based approaches have been proposed. Several works[9, 18, 51, 53] have outlined IR-based code representations for downstream optimizations. However, these are dependent on manual design choices and heuristics. Additionally, these representations usually need complex, resource intensive modeling techniques for each downstream task and might

increase the barrier to entry for new researchers. Working with self-supervised pre-trained models and using transfer learning for downstream tasks might help alleviate such shortcomings. This is our aim in this work.

To better represent source code/IRs, we believe modeling both syntax and semantics are equally important. And modeling each as separate modalities seems logical. However, representing source code as each such modality, and re-training from scratch for each target task adds complexity and increases resource requirements. Therefore, we propose an IR-based pre-trained encoder for performance optimizations. This allows us to remove dependency on individual programming languages and target optimizations on both CPUs and GPUs with the same pre-trained encoder.

Our Contributions. In this paper, we have proposed an IR-based self-supervised multi-modal pre-training approach (`MIREncoder`) with the aim of generating encodings/features for downstream tasks. Unlike prior code representations, our pipeline is completely self-supervised and only needs an LLVM IR as input for both pre-training and target optimization tasks. The IR statements in the input files are modeled to extract syntactic features during the pre-training process. This represents the first modality in our pre-training pipeline. The input IRs are also converted to multi-graphs that includes data-flow, control-flow, and call-flow information. This forms the second modality of our approach.

`MIREncoder` employs three pre-training tasks. The first modality, or IR statements are pre-trained on the task of **Masked Language Modeling** (MLM) with a Transformer based model. MLM is widely used in pre-training deep learning approaches with code or text generation capabilities. The second modality, or code graphs, are pre-trained with an auto-encoding task (**Graph Auto-Encoder**), where the aim is for a Graph Neural Network (GNN) based model to reconstruct the input graph. To the best of our knowledge, this study is the first to pre-train a multi-modal encoder using Transformers and GNNs to model individual modalities for parallel code. We also propose a new pre-training task to link the two modalities. We design a pre-training task to match the code graphs to the tokenized IRs (**IR-Graph Matching**). This allows our pre-trained model to better understand how the IR text translates to its corresponding graph, thus implicitly allowing the model to understand and link the syntactic, semantic, and structural aspects of the input IR.

We will show in later sections that the features/embeddings generated by our pre-trained model helps us match or outperform the state-of-the-art task specific approaches. Our `MIREncoder`-based embeddings lead to accuracy of upto ≈ 94% for CPU/GPU device mapping, speedups of upto 1.3×, 1.32×, ≈ 3× on thread coarsening, loop vectorization, and OpenMP paramter tuning tasks. Our predictions also reduce error rates by upto ≈ 40% and ≈ 70% over the state of the art for NUMA/Prefetcher optimizations, and tuning thread block sizes for CUDA code respectively.

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

- A multi-modal IR-based pre-training approach for source code representation.
- A novel pipeline that aims to i) model IRs as streams of lexical tokens with transformers, and ii) as multi-graphs with GNNs, to extract and understand syntactic, semantic, and structural features.
- A novel pre-training task, *IR-Graph Matching*, to link the two modalities and help the model relate syntactic, semantic, and structural features.
- Extensive experimental evaluations on *six* downstream tasks, including CPU/GPU device mapping, thread coarsening, loop vectorization, OpenMP parameter tuning, NUMA/ Prefetcher optimization, and tuning CUDA code with thread blocks, with superior results over state of the art.
- Analysis of the importance of each modality and the overheads of our pipeline.

2 Background

In this section, we briefly describe the topics relevant to this work.

2.1 Code Representations and Deep Learning

Recently, representation learning has been widely used for code modeling tasks. Several prior works have represented programs as a sequence of lexical tokens. However, this fails to capture program structure. To overcome this, syntax as well as semantics based representations have been proposed [2, 3, 9, 11, 21, 33, 41, 46, 51] that aim to extract and understand code structure as well.

`PROGRAML` [18] is such an IR-based code representation tool 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. We use `PROGRAML` to extract data, control, and call flow graphs from IRs.

2.2 Multimodal Deep Learning

Multi-modal learning relates information from multiple sources towards a common goal [37]. If a task can be represented in multiple ways, it can be assigned as multi-modal, with each representation defined as a unique modality. Multi-modal learning has been mostly applied to audio and video analysis, speech synthesis, and gesture recognition tasks [48]. For example, in image and video description tasks, the visual content and associated textual description can be considered different modalities of the same problem.

We take inspiration from these ideas and apply it to the task of code representation. A sequential and graphical code representation has been used to represent different modalities of the same piece of code. High-level embeddings obtained from each pre-trained modality are combined and associated to generate the feature space for downstream tasks.

Multi-modal Pre-trained Models. The remarkable success of pre-trained models in NLP has driven the development of multi-modal pre-trained model that learns implicit alignment between inputs of different modalities. These models are typically learned from bimodal data, such as pairs of language-image or pairs of language video, for example, `ViLBERT`[34]. Similarly, `VideoBERT`[49] learns from language-video data and is trained by video and text masked token prediction. With respect to pre-trained models targeting programming languages, `CodeBERT`[27] was trained on bimodal data with natural language and programming language pairs. Code comment and source code pairs were used for pre-training. However,

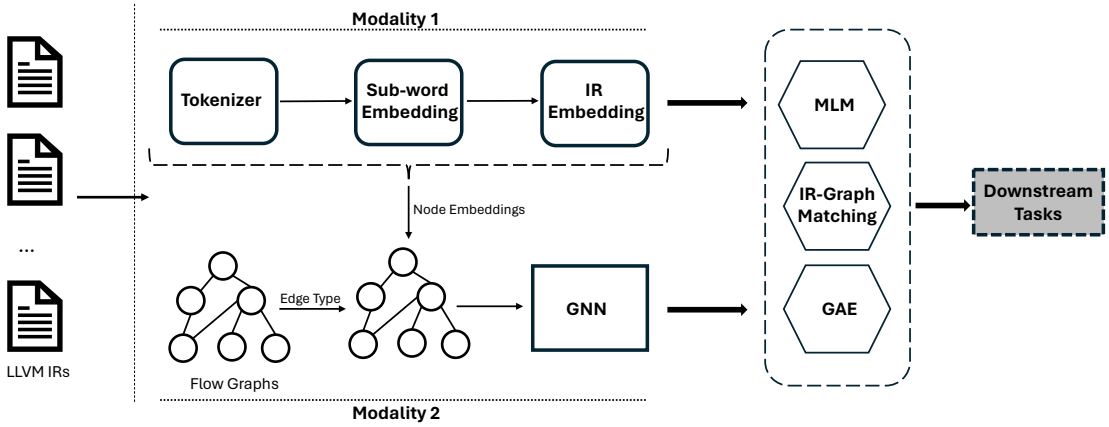

Figure 1: MIREncoder: Overview of our Multi-modal pre-training approach with two modalities using Masked Language Modeling (MLM), Graph Auto-Encoder(GAE), and IR-Graph Matching as pre-training tasks.

our work is different from these prior works, as we aim to only work with source code, and we consider two ways of representing code as separate modalities. Also, unlike prior pre-trained works, we only work with compilable code with a focus on generating features for performance optimization, rather than code generation.

3 MIREncoder

Most source-code based performance optimization tasks in HPC usually involve compilable languages such as C, C++, CUDA, and so on. A large number of these languages can be compiled and optimized using the LLVM infrastructure. LLVM IRs are a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes. It is fairly simple to extract IRs from source code such as C, C++. IRs generated from source code are usually devoid of most stylistic choices and redundant code. This is why we choose to work with IRs for performance optimizations. Figure 1 shows a high-level overview of our approach. For the first modality, we first tokenize the input IRs into meaningful "tokens" before they are mapped to an embedded dimension. Our approach then learns the embedding of the IR instructions after splitting them into sub-words. For the second modality, the IRs are first converted to dependence graphs that include in them data flow, control flow, and call flow information that represents the semantic information in the source code. These two modalities are then passed into the modeling pipeline either for pre-training or inference. The following paragraphs outline our pipeline.

3.1 Tokenization

Simply put, *tokenization* is the process of breaking down a piece of text into smaller units called tokens, and assigning a numerical value to each token. A deep learning (DL) model does not understand text or images in its raw form. It needs to be represented as numbers for the model to make sense from it. This is why tokenization is extremely important for such works. In this paper, our tokenization process follows the same approach taken while designing and training the `BERT`[23] model. However, the pre-trained `BERT` tokenizer readily available online is trained on natural language (NL). However, source code (IRs in our paper) is more structured than NL, and quite possibly has fewer "words". Thus, we had to train our tokenizer from scratch. We initially collect a large set of IRs by compiling programs in existing datasets into their LLVM IRs. For training the tokenizer, we have used C, C++, and CUDA code from `CodeNet`[39], `HPCorpus` [31], and `LS-CAT`[10]. We first define a set of special tokens to handle unknown inputs, and a token that will be used during Masked Language Modeling. 10,000 unique programs are randomly selected and compiled into LLVM IRs. These are then passed through a `WordPiece`[59] tokenizer, as done in `BERT`, and trained to generate a learned vocabulary. `BERT` uses a sequence length of 512. However, for the sake of simplicity and faster training, we limit the sequence length for each encoded IR statement to 64. Increasing the sequence length might improve results, but the aim of our work is to extract features from IRs, rather than have code generation capabilities. Thus, such an approach might be sufficient for performance optimization tasks, as we will show later.

Sample IR:

```
%"class.std::ios_base" = type { i32 (...)**, i64, i32, i32, i32, i64, i64,
%"struct.std::ios_base::_Iosarray"*, %"struct.std::ios_base::_Fnarray"*, %"class.std::locale"* }
```

Encoded IR:

```
[1, 9, 6, 114, 18, 108, 30, 30, 251, 41, 174, 6, 33, 273, 68, 124, 12, 18, 18, 18, 13, 14, 14, 16, 127, 16, 124, 16, 124, 16,
124, 16, 127, 16, 127, 16, 9, 6, 253, 18, 108, 30, 30, 251, 41, 174, 30, 30, 41, 2257, 6, 14, 16, 9, 6, 253, 18, 108, 30,
30, 251, 41, 174, 30, 30, 41, 2258, 6, 14, 16, 9, 6, 114, 18, 108, 30, 30, 237, 6, 14, 70, 2]
```

Decoded IR:

```
['[CLS]', '%', '"', 'class', '.', 'std', ':', ':', 'ios', '_', 'base', '"', '=', 'type', '{', 'i32', '(', '.', '.', '.', ')', '*', '*', ',',
'i64', ',', 'i32', ',', 'i32', ',', 'i32', ',', 'i64', ',', 'i64', ',', '%', '"', 'struct', '.', 'std', ':', ':', 'ios', '_', 'base', '*',
',', '%', '"', 'struct', '.', 'std', ':', ':', 'ios', '_', 'base', '::', '_', 'fnarray', '"', '*', '}', '[SEP]']
```

Figure 2: Example showing encoding and decoding with the MIREncoder tokenizer.

In Figure 2, we show an example of the tokenization process with our trained tokenizer. For this example, we select an IR statement from a file that was not used to train the tokenizer. We feed the statement to the tokenizer, which outputs a sequence of numbers (*input ids*). This is what a DL model will work with. To show that the encoding is correct, we decode the tokenized input ids to show that it is exactly the same as the given IR input, with a few minor

but important differences. As shown in Figure 2, the outputs are in array format, as the tokenizer decodes each input id individually. The array includes a '[CLS]' and a '[SEP]' token at each end. The '[CLS]' token is used to denote the class of the input, if applicable, and the '[SEP]' token is used to separate two statements in the same input. The upper case alphabets in the inputs have also been converted to lower case to make the sequences case-insensitive. If we remove the first and last tokens in the array, and join the elements, we end up with the same output as the input, which shows the success of our tokenizer training process.

3.2 Graph Generation and Pre-Processing

Several works ([2, 3, 9, 11, 21, 33, 41, 46]) have outlined that simply looking at source code as a stream of lexical tokens is not sufficient to represent code. Modeling IRs only as stream of tokens does not provide enough details about the structural properties of the program. Code structure can highlight dependencies in source code. It can show the flow of execution in source code, or can also show dependencies between variables. Given that such dependencies are sparse in nature, a graph seems to be an appropriate data structure to represent such structure and dependencies. The dependencies also highlight the meaning of a source code. The sequence of execution or the control flow, how the variables are dependent on each other or the data flow and the function call stack in a program are indicators of the underlying semantics of source code.

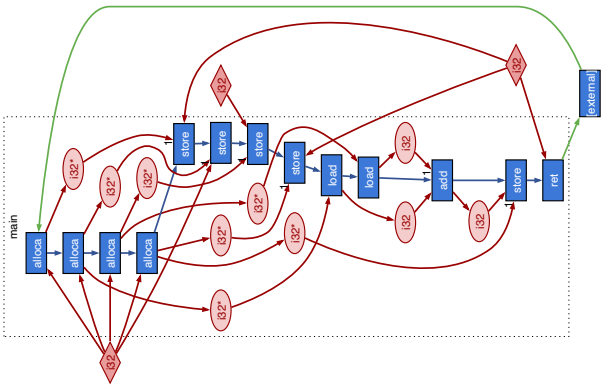

Figure 3: Example of code graphs used in this study. This is a graph for a simple program adding two numbers.

Prior literature [24, 26, 52] has used such structural and semantic information to good effect. We build on these ideas and work with graphs generated from IRs as the second modality. These graphs are generated with a tool called `PROGRAML`[18]. The generated multi-graphs contain data-flow, control-flow, and call-flow dependencies in them. During pre-training, these graphs allow our model to extract semantic and structural features from source code (IRs). This is necessary as code structure and semantics should dictate the performance of an application/kernel. An example of such a graph is shown in Figure 3.

The nodes in our generated graphs (example shown in Figure 3) contain IR statements. These form the node features in our graphs. Node features are used by Graph Neural Networks (GNNs) in forward and backward propagation during training. However, DL models cannot use such statements directly. Therefore, we use the trained tokenizer described in Section 3.1 to convert the IR statements into sequence of numbers. These become the node features and are used in the pre-training process by the GNN layers.

3.3 Pre-Training MIREncoder

In this section we outline the pre-training process of `MIREncoder`. The quality of a pre-trained model usually depends on the pre-training tasks considered. For this work, we have used three pre-training tasks; one task that targets each modality, and another one that is used to explicitly link together the two modalities. Namely, the pre-training tasks are *Masked Language Modeling*, *Graph Auto-Encoding*, and *IR Code-Graph Matching*.

3.3.1 Masked Language Modeling. Masked Language Modeling (MLM) is a widely used pre-training task in natural language based pre-trained models. It is also commonly used as a pre-training task in studies working with programming languages such as `CodeBERT` [27].

MLM for this paper can be defined as follows. Given an IR statement c as input, we select a random set of positions m^c that will be masked out; i.e. replaced with the '[MASK'] token. Following ideas presented in [23, 27], we mask out 15% of the input. The task in this pre-training step is for the model to successfully predict the masked out words from the adjoining context. This is a self-supervised approach as the model is expected to produce the correct outputs without any explicit labels. Throughout the training process, the model is updated based on the difference between the predicted words and the actual words in the statements.

However, it is worth noting that the '[MASK]' token does not appear during the downstream tasks. To overcome this, as done in [23], we perform the following steps:

- Select 15% of the token positions at random.
- Randomly replace 80% of the selected positions with the '[MASK]' token.
- Replace 10% of the selected positions with a random token.
- Keep the token unchanged for the remaining cases.

These steps help the model learn the meaning of a word in the context of a statement, and not assign a single meaning to a word. Also, not including the '[MASK]' token in each statement during pre-training ensures that the model does not always expect that token. For this pre-training task, we use transformer layers with attention mechanism for improved training.

3.3.2 Graph Auto-Encoding. Graph Auto-Encoders (GAEs) like traditional auto-encoders also aim to reconstruct the given inputs. The aim of this pre-training task is for the model to produce a learned low-dimensional embedded representation from the IR graphs during the downstream tasks. During pre-training, our model setup follows the widely used *encode-code-decode* setup. An input graph is first fed through GNN layers (*Graph Convolution Layers* or *GCN*) to produce node embeddings in a two-dimensional latent space. This forms the *encoder* part of the network. In the *decoder* part of the network, the aim is to reconstruct the graph from the low-dimensional encoded form. The aim is not to reconstruct the original nodes, but to reconstruct the adjacency matrix identical to the input graph

through an inner product between latent variables in order to understand the structure of the graphs.

Now the multi-graphs used in this study have three sub-graphs in them denoting control-flow graphs, data-flow graphs, and call-flow graphs. However, it is quite difficult to auto-encode graphs with multiple edge types. Therefore, we tweak the training process slightly by extracting each sub-graph from the IR multi-graph, and train the auto-encoder for each of the three sub-graphs. But, we do not train the model thrice. The modeling and the loss calculation phases are updated to work with the node features and adjacency matrices of each sub-graph. The loss is back-propagated as an aggregation of the difference in graph reconstruction of each sub-graph. There are two main benefits to this: i) calculating the loss and back-propagating over the whole graph instead of each sub-graph allows the model to improve its learning over the whole graph and enables it to implicitly learn the relations between the three types of semantics in the graphs (control-flow, data-flow, call-flow), ii) it improves overall training time when compared to training three separate GAEs, one for each sub-graph.

3.3.3 IR-Graph Matching. Here, we propose a novel pre-training task IR-Graph Matching to link the two modalities together. The modalities considered in this paper have different data structures, one being a sequence of tokens, the other being a graph. Intuitively, it might be difficult for the model to understand how these two modalities are linked together, and by extension, difficult to link the syntax and structure.

Therefore, we propose this pre-training task where the aim is for the model to correctly identify if the code sequence and the code graphs are from the same IR source. We setup this as a binary classification task, where the inputs are the code sequences (S) and the code graphs (G). Positive and negative samples are automatically generated as data pairs to train the model. Positive samples are those where S and G are from the same IR, while the negative samples are those where the graphs and sequences are from different IRs. Negative samples are selected in 50% cases by randomly selecting a different IR from the dataset. The code graph of the negative sample is paired with the code sequence to create the negative data pair.

As outlined in Section 3.3.1, the Masked Language Modeling task is performed on IR statements. However, in this task, we need to work with whole files to match text in IR files to the corresponding graphs. Although embedding an IR statement/instruction to a sequence of length 64 might work, embedding a complete file with a large number of statements to a sequence of length 64 will not provide enough information to the model. Therefore, we embed each statement in the file, and then aggregate all the vectors. The aggregated input and the generated code graph with the embedded node features (Section 3.2) are then trained together as a binary classification problem. The transformer layers used in Section 3.3.1 and the GCN layers used in Section 3.3.2 are reused to model the code sequences and the code graphs. Their outputs are concatenated and passed through linear layers with binary cross-entropy used for the loss calculations.

4 Experiments

In this section, we outline the experiments undertaken to show the strength of our approach. We test our pre-trained model on *six* downstream tasks different from each other. For each task, we work with metrics used in prior works for evaluation. Experimental setup and evaluation metrics are outlined in more detail in the corresponding sections.

A few things, however, are common for all experiments. For each downstream task, the pre-trained model is *not* fine-tuned. The pre-trained model is set to inference mode to generate embeddings for input IRs. A few trainable linear/MLP layers are added to the pre-trained model to perform task specific training. For downstream tasks, only these final layers are trained which substantially reduces the optimization/tuning overhead. In the following sections, we outline each downstream optimization task performed in this paper, and compare and contrast our results with the state of the art.

4.1 Heterogeneous Device Mapping

Grewe et al. [28] proposed the device mapping task to map `OpenCL` kernels to the CPU or GPU. This task has been widely used[9, 18, 19, 51, 53] to evaluate the performance of code representations. We also use this task to evaluate the effectiveness of our approach and compare against the state-of-the-art results.

Dataset. We use the dataset published by Ben-Nun et al. [9] for this experiment. It has 256 unique `OpenCL` kernels from seven benchmark suites comprising of AMD SDK[6], NPB[8], NVIDIA SDK[17], Parboil[47], Polybench[38], Rodinia[14, 15] and SHOC[22]. 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.

Baseline. We have compared the results of our approach with prior works with the same dataset. Prior evaluations were presented in terms of accuracy and performance improvements (speedups). We have also adhered to these metrics. For analysing speedups, we use the same static mapping baseline proposed in [53].

Results. We use our pre-trained model to encode the available IRs, and perform the classification experiments. The `MIREncoder` pipeline is first used to embed each IR statement in a file. The generated embeddings are then aggregated to encode the first modality. For the second modality, the IRs are first converted to graphs as outlined in Section 3.2, and are fed through the Graph Auto-Encoder (GAE) layers to encode the graphs. These two sets of embeddings (sequences of vectors) are passed through three linear/MLP layers to train and validate the model. As done in prior works, we also add transfer and workgroup sizes from the dataset to the feature set before passing the feature set onto the linear layers. Following techniques used before [18, 51, 53], we have used ten-fold stratified cross-validation to evaluate our results.

Our experimental setup leads to state-of-the-art results in identifying the correct device. We achieve accuracy of 93.7% and F1-score of 0.94 in identifying the best device on the NVIDIA GPU. On the AMD GPU, we achieve accuracy and F1-score of 93.6% and 0.92. We see that our approach is better or equivalent in all cases compared to prior works in literature. The accuracies are shown in Table 1, and the numbers in parenthesis shows the improvement in accuracy by using `MIREncoder` over prior works.

The model predictions also lead to significant performance improvement over static mappings. On the NVIDIA 970 system, our approach leads to speedups of 1.28× compared to oracle speedups

Table 1: Accuracy: (CPU/GPU) device mapping.

State-of-the-art	NVIDIA GPU (%)*	AMD GPU (%)*
Grewe et al. [28]	74.56 (25.67)	70.29 (33.16)
DeepTune [19]	80.88 (15.85)	83.24 (12.44)
inst2Vec [9]	82.65 (13.37)	82.35 (13.66)
PROGRAML [18]	80 (17.13)	86.6 (8.08)
IR2Vec [53]	89.68 (4.48)	92.82 (0.84)
Perfograph [51]	90 (4.47)	**94** (-0.42)
MIREncoder (ours)	**93.7**	93.6

* Numbers in parenthesis are percentage improvements in accuracy over prior works.

Table 2: Thread Coarsening Factors: Speedups obtained by prior works. Best speedups are highlighted in bold.

Device	DT	NCC	IV	PG	ME
AMD Radeon HD 5900	1.1	1.29	1.24	1.19	**1.29**
AMD Tahiti 7970	1.05	1.07	1.30	1.14	**1.30**
NVIDIA GTX 480	1.1	0.97	1.25	1.03	**1.26**
NVIDIA Tesla K20c	0.99	1.01	**1.16**	1.01	**1.16**
Average	1.06	1.09	1.23	1.09	**1.24**

DT=DeepTune, **NCC**=inst2vec, **IV**=IR2Vec, **PG**=Perfograph, **ME**=MIREncoder (ours)

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. On the AMD Tahiti system, our predictions lead to speedups of 2.24× versus oracle speedups of 2.39×.

4.2 Thread Coarsening

Thread coarsening[54] is used to increase the work done by a single thread by fusing two or more concurrent threads. Thread coarsening factor (TCF) corresponds to the number of threads that can be fused together. Selection of an optimal TCF can lead to substantial improvement[36] in performance on GPU devices and a naive coarsening could lead to slowdown. Due to differences in architectural characteristics across devices, a TCF that gives the best speedup on one GPU might show degraded performance on another GPU[35, 45]. As an example, *nbody* kernel has a higher degree of Instruction Level Parallelism and can be better exploited by VLIW-based AMD Radeon than SIMD-based AMD Tahiti[35].

Dataset. In this experiment, we follow the experimental setup proposed in [35] and reused in [53] to predict the best thread coarsening factor from {1, 2, 4, 8, 16, 32}. We use the dataset provided in [9], which consists of 68 data points from 17 OpenCL kernels on 4 different GPUs, namely AMD Radeon 5900, AMD Tahiti 7970, NVIDIA GTX 480, and NVIDIA Tesla K20c. These include kernels from AMD SDK[6], NVIDIA SDK[17] and Parboil[47] benchmarks.

Baseline. As done in prior works[51, 53], we evaluate the predictions from our model in terms of performance improvement/speedups over default coarsening behavior. The results from our approach has been compared against prior works on this dataset.

Results. For this experiment, we pass the input IRs through our pre-trained encoder as before to generate the embeddings. The embeddings are then passed through linear layers to train and validate the best thread coarsening factors. Similar to prior works on this task, we also perform leave-one-out cross validation and report the geometric mean speedups across all folds in Table 2. We observe that our approach performs better in all cases. In Table 2, speedups are presented for each device included in the dataset.

Table 3: Speedups: Improvements in runtime over LLVM vectorization. LLVM vectorization speedups are always 1.0×.

LLVM	Neurovectorizer	MIREncoder (ours)
1.0×	1.22×	**1.32×**

4.3 Loop Vectorization

Modern compilers can automatically detect when loops should be vectorized so that multiple iterations of the loop can be performed together. When compilers vectorize loops, it must determine the number of instructions to pack together and the interleaving level (stride). This task was proposed in [29] as a potential candidate for DL-based optimization. Modern compilers allow users to select and define the vectorization factor (VF) and interleave factor (IF) to control the loop vectorization process. However, manually evaluating and testing all possible combinations might not be feasible, especially for a large number of applications. To this end, we propose a `MIREncoder`-based static loop vectorizer.

Dataset. We define a search space based on ideas in [29] by considering pairs of VF and IF and execute them to create a dataset. Their definitions are given below in Equation 1,

$$\begin{aligned} VF &\in [2^0,\ 2^1,\ \ldots MAX\_VF], \\ IF &\in [2^0,\ 2^1,\ \ldots MAX\_IF], \end{aligned} \tag{1}$$

where we set *MAX_VF* and *MAX_IF* to 64 and 16 for the architecture under test (Intel Skylake). We reuse the set of kernels collected in [29] and execute them with each (VF, IF) pair to create a dataset of kernels, (VF, IF) pairs, and their runtimes. The training and testing set were defined separately in [29] and we follow the same setup here as well. Overall, we collect more than $273K$ samples for training. We label the kernels with the best (VF, IR) pair by selecting the vectorization/interleave factor with the fastest runtime. For the test set, we perform the same steps to create the test set.

Baseline. For this experiment, we select the default LLVM `Loop Vectorizer` as a baseline and evaluate the predicted performance with respect to this. We compare our work with `Neurovectorizer`[29]. This paper first proposed this task as suitable for DL-based tuning. They used `inst2vec`[9] embeddings with reinforcement learning for their experiments.

Results. We follow the same steps as before in this experiment as well. We pass the IRs through the pre-trained model to generate the embeddings. The embeddings are then passed through the trainable MLP layers to train and test the model. Both vectorization factor and interleave factor can be varied during compilation. Therefore, we depend only on the compiled IR for training and testing. We train our model on the training data collected on our Intel Skylake server, and test it on the collected test set. From our experiments we see that `MIREncoder`-based vectorization leads to mean speedups of $\approx$ **1.32**× over LLVM vectorization heuristics. We repeat the same experiments as done in [29] on the same Skylake server and observe

Table 4: Search space for tuning OpenMP parameters.

Parameter Name	Parameter Values
Power Limits	75W, 100W, 120W, 150W
Number of threads	1, 4, 8, 16, 32, 64
Scheduling Policy	STATIC, DYNAMIC, GUIDED
Chunk Sizes	1, 8, 32, 64, 128, 256, 512

that using `Neurovectorizer` leads to speedups of ≈ **1.22**× across all kernels in the test set (Table 3).

4.4 Tuning OpenMP Runtime Configurations

OpenMP is one of the most widely used shared memory programming models. It is mostly used to parallelize sequential code by inserting *pragmas*. Dutta et al. in [26], proposed a GNN-based tuner (PnP Tuner) for identifying the best OpenMP parameters for improving performance. They also used *power limits* to increase the size of the search space and evaluate the impact that limiting power has on OpenMP applications. We build on ideas presented in that paper to set up our own tuner based on MIREncoder.

Dataset. As in [26], we work with 25 applications from the Polybench benchmark suite. We define the search space as done in [26] (Table 4) with 504 configurations. We also modify the data sizes used to run these applications by changing compile time options provided by the benchmark suite. We use two input sizes for the purposes of evaluation. For each application, input size, and parameter set (power limit, # of threads, schedule, chunk) we compile and execute the application to collect the runtimes and generate a dataset of 25200 samples. We also collect the runtimes for each of these applications when run with default OpenMP settings (all threads, static scheduling, compiler defined chunk sizes) at *Thermal Design Power* (*TDP*). We collect this data on a 64-core Intel Skylake system with a TDP of 150*W*.

Baseline. The metric of choice for evaluating the performance of our MIREncoder-based tuner is speedups as done in [26]. We calculate the speedups with the default OpenMP configurations at *TDP* as the baseline and compare the performance of our predicted configurations with those predicted by the PnP tuner, the current state-of-the-art for this experiment. In fact, PnP tuner also works with graphs generated from IRs. Their approach, like ours, also aims to model control and data flow in a program with the help of GNNs. PnP tuner uses RGCN (Relational Graph Convolutional Networks) as the GNNs of choice.

Results. Following the approach in [26], for this set of experiments we consider application speedups instead of performance improvement of individual OpenMP loops. Also, the OpenMP parameters were modified at runtime. Thereby, all OpenMP loops in an application were run with the same set of parameters. The modeling process used in this experiment is also similar to the ones used in previous sections. For validation, we perform *leave-one-out* validation as done in [26]. Each application is assigned to the test set, while all other applications are assigned to the training set. We repeat this for all applications in the dataset. Based on our experiments, we find that the tuner designed with MIREncoder embeddings has better or equivalent performance to PnP tuner. It is able to identify configurations that lead to faster code execution in most cases,

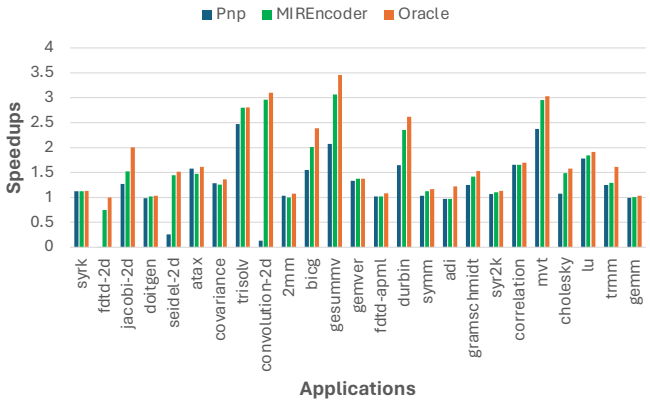

Figure 4: Auto-tuning power limits and runtime parameters for OpenMP applications (Higher is better).

sometimes improving runtime performance by ≈ 3×. Identifying such configurations are often non-intuitive, and identifying such simple runtime parameters can help improve the performance of parallel applications. Across 25 applications, MIREncoder embeddings helps our tuner reach near-optimum performance (> 0.9× of oracle runtimes) in 16 cases out of 25. In comparison, PnP tuner only reaches such performance in 11 out of 25 cases. Additionally, the configurations predicted by our tuner lead to slowdowns in only one case. In contrast, PnP leads to slowdowns in six cases. Overall, the MIREncoder-based tuner outperforms the PnP tuner in 22 out of 25 cases.

4.5 Optimizing NUMA/Prefetcher Parameters

TehraniJamsaz et al. in [52] proposed a novel GNN-based LLVM IR modeling technique for optimizing NUMA (Non-Uniform Memory Access) and Prefetcher configurations. In particular, this work built on top of prior works[43] to explore the impact of various cache prefetching options along with NUMA-related hardware parameters such as number of threads, degree of NUMA node, thread mapping and page mapping. The authors used graph embeddings generated from LLVM IRs to statically map each kernel to the best NUMA/prefetcher configuration.

Dataset. In [52], the authors used data from 57 parallel kernels from Rodinia[14, 15], NAS Parallel Benchmarks[8, 44], CLOMP[12], and LULESH[32]. The data was collected on Intel SandyBridge and Intel Skylake processors on a search space with 288 and 320 configurations respectively. This dataset was pared down to 13 configurations as the authors found that 99% of the performance gains were obtained using these 13 configurations. To increase the quality of their code modeling and improve results, TehraniJamsaz et al. augmented the dataset by re-compiling the kernels in the dataset with 1000 different compiler sequences. We use this collected dataset to further test the strength of our approach.

Baseline. In this study, we are using a pre-trained model to generate the embeddings for each IR. In addition to testing the quality of optimizations made by our MIREncoder embeddings, we also use this experiment to highlight reduced data requirements when a pre-trained model is used to generate features. *Transfer*

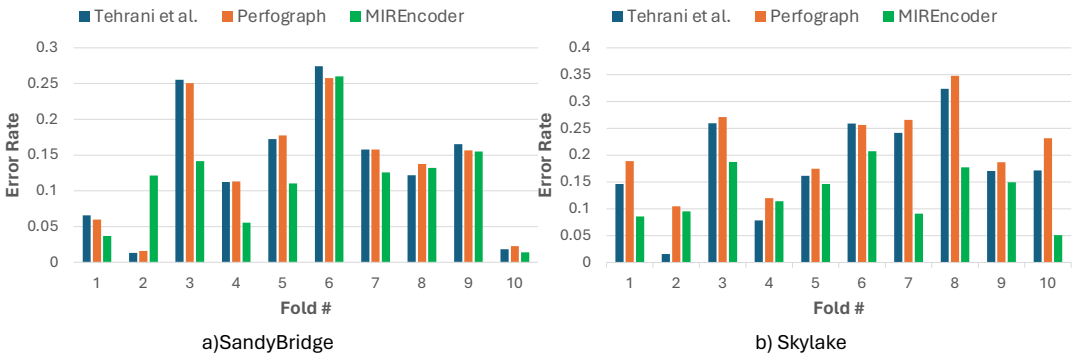

Figure 5: Error rates for predicting NUMA and prefetcher configurations for parallel code regions (lower is better).

learning allows us to achieve this as `MIREncoder` generates learned embeddings, thus implicitly transferring its knowledge to the tuner. Therefore, during training we only use ≈ 5% of the complete dataset for training. During validation and testing, the authors in [51, 52] used 10-fold validation. We also use the same folds for our tests and compare the results from `MIREncoder` with the state of the art[51, 52] in this dataset. As with prior works on this task, we also use error rate (relative difference between best and predicted performance) as the evaluation metric.

Results. To perform 10-fold validation, we first separate the validation set from the training set by assigning the kernels specified in each fold to the validation set. During training, we only select ≈ 5% of the IRs in the dataset at random for the kernels in the training set. However, we validate on all IRs corresponding to the kernels in the validation set. Training with such reduced data also produces good results as we can leverage transfer learning from our pre-trained model to generate embeddings for the IRs in the training set. For both SandyBridge and Skylake, we outperform [52] in 8 out of 10 folds (Figure 5). The modeling for this experiment only uses simple MLP layers in contrast to [51, 52], which trains resource intensive GNNs for each experiment. Overall, across 10 folds, `MIREncoder` embeddings help reduce performance error rates by ≈ 15% (SandyBridge) and ≈ 29% (Skylake) over [52]. `MIREncoder` embeddings outperform `Perfograph`[51] in 8 out of 10 folds for SandyBridge improving error rates by ≈ 14%. It outperforms `Perfograph` in all cases for Skylake, improving error rates by ≈ 40%.

4.6 Tuning Thread Blocks for CUDA Programs

So far the auto-tuning experiments have targeted programs written in C and C++. However, state-of-the-art GPUs have contributed immensely to performance improvement of HPC workloads and CUDA is often the language of choice for programming such GPUs, specifically NVIDIA GPUs. With this in mind, we have tried to optimize the performance of CUDA kernels in this section. As in the prior sections, we work with a previously published dataset and use a `MIREncoder` based tuner to identify the best parameters to run CUDA kernels.

Dataset. To address the lack of large scale datasets suitable for machine learning based optimizations of CUDA kernels, Bjertnes et al. published the `LS-CAT`[10] dataset with 19,683 CUDA kernels. They also open source scripts to modify the input matrix sizes, and thread blocks used to execute these kernels. We compile and run these CUDA kernels with the matrix sizes and thread blocks shown in Table 5 to collect a dataset with more than 2.7 million samples on an NVIDIA A100 GPU. From the collected dataset, we identify the minimum runtime of each kernel and input matrix. The block size corresponding to the fastest runtime is then selected as the best configuration. This processed and labelled data is then used to train a simple MLP model on the `MIREncoder` embeddings to predict the best configuration for a CUDA kernel and input matrix unknown to the model.

Table 5: Parameters Modified for CUDA kernels.

Param. Name	Param. Values
Matrix Size	240, 496, 784, 1016, 1232, 1680, 2024
Block Sizes	(8,8), (16,16), (24,24), (32,32), (1,64), (1,128), (1,192), (1,256), (1,320), (1,384), (1,448), (1,512), (1,576), (1,640), (1,704), (1,768), (1,832), (1,896), (1,960), (1,1024)

Baseline. To the best of our knowledge, this study is one of the first works to perform optimizations using this dataset for CUDA code. To evaluate our `MIREncoder` representation, we use the embeddings from three prior works (`IR2Vec`[53], `PROGRAML`[18], `Perfograph`[51]), and adapt the modeling techniques specified in those papers to the best of our ability for this task. We follow the same strategy used in [52] and Section 4.5 and use error rates (relative difference between best and predicted performance) as a metric to present and compare the results for this section.

Results. The `LS-CAT` dataset does not have a designated test set. Therefore, we perform 10-fold validation as done in prior works[18, 52] and sections. We build four ML-based auto-tuners to model `MIREncoder`, `Perfograph`, `PROGRAML`, and `IR2Vec` embeddings. The `IR2Vec`, `PROGRAML`, and `Perfograph` embeddings were modeled with the techniques outlined in the respective studies. The `MIREncoder` based tuner uses only simple MLP layers. As shown in Figure 6, the `MIREncoder` embeddings outperform the state-of-the-art even with a very simple network. Our tuner produces better results than the `IR2Vec` and `Perfograph`-based tuners in 8 and 9 folds out of 10. It outperforms `PROGRAML` in all folds. Across all folds,

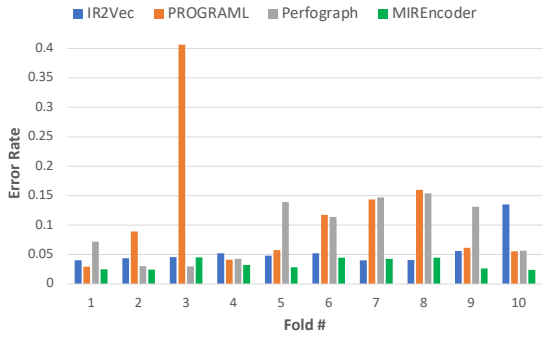

Figure 6: Error rates for predicting thread blocks for CUDA kernels (lower is better).

Table 6: Ablation Studies (CS1): Heterogeneous device mapping. Change in accuracy when one modality is removed.

State-of-the-art	NVIDIA(%)	AMD(%)
Grewe et al. [28]	74.56	70.29
DeepTune [19]	80.88	83.24
inst2Vec [9]	82.65	82.35
PROGRAML [18]	80	86.6
IR2Vec [53]	89.68	92.82
Perfograph [51]	90	94
MIREncoder (only IR text)	59.1	79.7
MIREncoder (only IR graphs)	86.2	88.5
MIREncoder	93.7	93.6

our predictions reduce error rate over `IR2Vec`, `Perfograph`, and `PROGRAML` by ≈ 39%, ≈ 63% and ≈ 70% respectively.

4.7 Observation and Analysis

In this section, we outline and analyse the merits of our approach. We primarily hope to show the importance of each modality to our pre-training pipeline. We will also show how using our approach reduces the overheads associated with deep learning based performance optimization.

Ablation Studies. Ablation studies are commonly used in deep learning to highlight the importance of individual components of the modeling process. Here, we hope to highlight the impact of each modality on the modeling process. We first remove the modules associated with each modality from the pipeline and pre-train the uni-modal models from scratch. However, for each uni-modal model, we only train it on one pre-training task as each pre-training task was designed with a modality in mind. For example, a *Masked Language Modeling* pre-training task would not be appropriate for the code graph modality. And the *IR-Graph Matching* task is dependent on both modalities being a part of the pre-training process. With this setup, we pre-train our uni-modal models and follow the same experimental setups as before. The uni-modal pre-trained models are tested on three tasks from the previous sections, namely heterogeneous device mapping (Section 4.1), thread coarsening (Section 4.2), and loop vectorization (Section 4.3).

Table 7: Ablation Studies (CS2): Thread Coarsening Factors: Changes in speedups when one modality is removed.

Device	DT	NCC	IV	PG	ME(T)	ME(G)	ME
AMD Radeon HD 5900	1.1	1.29	1.24	1.19	1.24	1.13	1.29
AMD Tahiti 7970	1.05	1.07	1.30	1.14	1.25	1.15	1.30
NVIDIA GTX 480	1.1	0.97	1.25	1.03	1.19	1.09	1.26
NVIDIA Tesla K20c	0.99	1.01	1.16	1.01	1.07	1.08	1.16
Average	1.06	1.09	1.23	1.09	1.18	1.11	1.24

DT=DeepTune, **NCC**=inst2vec, **IV**=IR2Vec, **PG**=Perfograph, **ME**=MIREncoder, **T**=Text Only, **G**=Graph Only

Table 8: Ablation Studies (CS3): Speedups over LLVM vectorization when individual modalities are used.

LLVM	Neurovec.	ME (T)	ME (G)	ME
1.0×	1.22×	1.01×	1.18×	1.32×

ME=MIREncoder, **Neurovec**=Neurovectorizer, **T**=Text Only, **G**=Graphs Only

Case Study 1 (CS1): Each experiment shows that our pre-training is highly dependent on each modality. For device mapping, the quality of the predictions fall significantly when modality 2 (code graphs) is not included. When modality 1 (IR text) is removed, the performance drops, but less drastically. When we only pre-train with modality 1, performance drops by ≈ 37% and ≈ 14% for the NVIDIA and the AMD GPUs, whereas performance drops by ≈ 8% and ≈ 4% when only code graphs are used for pre-training (Table 6). The higher dependence on the code graphs is expected as code semantics dictate which device is chosen as the best one for some of the kernels in this dataset. For example, the *makea* kernel from the `CG` application in NPB[8], has a faster runtime on the GPU with a smaller input size, whereas it is mapped to the CPU when run with larger inputs. This behavior can be due to the presence of a number of function calls inside the parallel kernel. Such semantic details might be difficult for an NLP-style model to understand. However, a graph that embeds such dependencies as edges between nodes can help highlight such semantic information to the model.

Case Study 2 (CS2): When predicting the thread coarsening factors, we see that not including the code graphs has a smaller impact on thread coarsening factors than device mapping. Moreover, using only the code graphs leads to a bigger drop in application performance than when using both modalities. We see that performance drops by 5% when the code graphs are not used, whereas performance drops by 11.7% when only code graphs are used (Table 7).

Case Study 3 (CS3): We also test how unimodality impacts the performance of loop vectorization. Loop vectorization is an important compiler optimization for modern processors. For this set of experiments as well, we see that removing a modality impacts the performance of the predictions (Table 8). Using IRs only in textual format the performance drops by ≈ 30%, and when we use only the code graphs as a modality the performance of our vectorizer drops by ≈ 12%.

Table 9: Slowdowns over `MIREncoder` (no FT) wall times.

Process	ME (*w/o FT*)	PnP (GNNs)	ME (*w. FT*)
Training	1×	37×	238×
Inference	1×	1.8×	1×

FT=Fine-tuning, **ME**=MIREncoder, **w/o**=*without*, **w.**=*with*

Analyzing Overheads. Most advanced DL-based works usually have significant training and inference overheads. We use the experiment in Section 4.4 as a template to evaluate the overhead of our approach. We first train and test the `MIREncoder`-based tuner and `PnP` tuner[26] from Section 4.4 and capture the wall times. The `PnP` tuner is a GNN based code modeling approach that first proposed this downstream task. Across all experiments, to reduce overhead, we simply generate embeddings from the pre-trained model instead of fine-tuning it. `PnP` on the other hand needs to train GNNs for each experiment. Compared to our `MIREncoder`-based tuner, which only trains a few MLP layers, training and tesing a GNN based model is much more expensive as shown in Table 9.

Most studies working with pre-trained models usually suggest fine-tuning the pre-trained models for downstream tasks. However, in this work, we *do not* fine-tune our pre-trained model for downstream tasks. The embeddings generated by `MIREncoder` are good enough to be used with simple shallow networks. To show this, we perform a set of tests with two setups: i) we use our regular set up where we *do* set the pre-training model to inference mode and use only the final MLP layers for training and testing, and ii) we *do not* set the pre-training model to inference mode, and use the complete network to fine-tune for downstream tasks. We observe that for the experiment in Section 4.4, performance (speedups) improves $< 5\%$ when we fine-tune. However, the training time balloons by 238×. This is a significant increase in overhead for fairly marginal gains. We avoid this overhead by *not* fine-tuning, but simply generating the embeddings to achieve good results as shown in Section 4. Such overheads are seen even for our relatively small model with 22 million parameters. Recently, large language models, with billions of parameters, have been proposed[20] for addressing compiler optimizations. This would increase the training time exponentially, especially when computational resources are limited. Thus, new innovative techniques, like the one proposed in this paper, is necessary to reduce overheads and large-scale resource dependence. Multi-modality allows us to work with a small model and helps offset the loss in learning when a small uni-modal model is used.

5 Related Works

This paper proposes a new pre-trained multi-modal code representation technique for LLVM IRs. For most source code based optimization tasks, analyzing code can provide pointers to the pertinent optimizations. In fact, most compiler optimizations are code dependent. Therefore, a suitable code representation technique is also essential for using deep learning (DL) to make optimization decisions in HPC. To this end, several code representations have been proposed [3, 4, 9, 11, 18, 21, 41, 53], which have been used to good effect for optimization tasks such as CPU/GPU device mapping, thread coarsening factor, loop vectorization, etc. to name a few. Preliminary works in this field such as [3–5], focused more on lexical tokens which often fails to capture code semantics. The next generation of representational learning works[9, 18, 24, 25, 51, 53] leverage LLVM IRs to make semantic features available to DL models. However, the embeddings generated by these often require advanced modeling techniques such as GNNs for each individual task. In contrast, for downstream tasks, our approach can leverage transfer learning to generate learned embeddings that can be easily modeled with simple MLP layers to get better results than these.

An alternative to DL-based auto-tuning is to use non-neural network based machine learning approaches. Several works have used ML for a variety of tasks. [40, 55] propose machine learning based approaches for auto-tuning `OpenMP` applications. Artemis[56] is another work that performs automatic parameter tuning using machine learning. `ytopt`[57, 58], `BLISS`[42] are examples of learning-based tuners that employ Bayesian optimization for online tuning tasks. These approaches are often domain or application specific. Although often faster than search-based alternatives, these do need multiple code executions to identify good performing parameters.

Studies highlighted so far in this section were all proposed as means to improve upon traditional search based auto-tuning. Works such as `ActiveHarmony`[50], `OpenTuner`[7] have leveraged several search space optimization techniques to reduce the auto-tuning overhead compared to *brute-force tuning*. These optimization techniques include Hillclimbers, random search, Nelder-Mead, and many more. However, due to their sampling overhead, works such as `ytopt` and `BLISS` were proposed to reduce tuning overhead.

DL-based approaches, including ours, further help alleviate such overhead by making predictions without having to execute applications. This helps with configuring commonly used parameters across applications, without having to devote significant resources to the tuning process.

6 Discussion

In this work, we have proposed a pre-trained multi-modal encoder for IRs with source code based performance optimizations in mind. Such an approach enables a model to understand syntactic, semantic and structural characteristics of source code. Prior works in this domain often depend only on NLP-style stylistic choices or compiler based code semantics and might require advanced modeling techniques with significant overheads.

Not only do our embeddings help reduce overhead on downstream tasks (Section 4.7), our pre-trained model is itself much smaller in scale than the latest pre-trained models in literature. Very large models, such as LLMs, often have billions/trillions of parameters. This makes training and fine-tuning them quite expensive, often requiring multiple state-of-the-art GPUs. Our pre-trained multi-modal model on the other hand, only consists of 22 million parameters, and can be easily trained using a single GPU. However, most very large models have text or image generation capabilities; our model does not. This is by design as the aim of this work is to simplify and speed up the process of deep learning based performance optimization in HPC.

Moreover, for downstream tasks, we do not need to fine-tune the pre-trained model as is often necessary for larger models. We simply put the pre-trained model in inference mode, and output the embeddings of an input LLVM IR. Transfer learning allows us

to do this and still achieve good results across multiple languages (C, C++, CUDA) and programming models (OpenCL, OpenMP). Because the pre-trained model has already been trained to understand and relate code syntax, semantics and structure, during downstream optimization tasks, the pre-trained model can leverage its prior knowledge to generate good quality embeddings. This also allows us to reduce data requirement while training DL models, as shown in Section 4.5, where we train our model with only 5% of the data that the state of the art[51, 52] had been trained on.

Additionally, our pipeline is modular by design. This can inspire future research on how each modality can be represented. For example, we could replace the graphs used in this work by other graphical representations such as ASTs, Perfograph, Graph2Par[16] and evaluate their impact. We hope to do this in future.

7 Conclusion and Future Works

This paper proposes MIREncoder, a multi-modal pre-training approach to encode/embed LLVM IRs for easy use by deep learning based models targeting performance optimizations in HPC. Our pre-trained encoder will allow researchers to focus more on adapting deep learning for HPC optimization problems instead of focusing on how it can be done. Moreover, as seen widely in literature, it is often possible to re-train existing pre-trained models for multiple domains. With this in mind, our model has been designed to be smaller in scale compared to existing pre-trained models. This would allow further research on such topics, and would not make researchers completely dependent on high-end and large-scale resources as is the case with very large models. Our aim with this paper was to propose a pre-training pipeline for HPC that would be small-scale. We helped alleviate the loss in learning from using a smaller model by introducing multi-modality to help our model better understand code "meaning". Our experimental results and further analysis support our claims of better performance with reduced overheads. Furthermore, our pre-trained model could easily be used in conjunction with online auto-tuners to help aid the search process. We hope to investigate this in future.

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.

References

[1] Jordi Alcaraz, Ali TehraniJamsaz, Akash Dutta, Anna Sikora, Ali Jannesari, Joan Sorribes, and Eduardo Cesar. 2023. Predicting number of threads using balanced datasets for openmp regions. *Computing* 105, 5 (2023), 999–1017.

[2] 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.

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

[4] 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.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

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

[7] 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.

[8] 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.

[9] 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).

[10] Lars Bjertnes, Jacob O Tørring, and Anne C Elster. 2021. LS-CAT: a large-scale CUDA AutoTuning dataset. In *2021 International Conference on Applied Artificial Intelligence (ICAPAI)*. IEEE, 1–6.

[11] 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.

[12] Greg Bronevetsky, John Gyllenhaal, and Bronis R De Supinski. 2008. CLOMP: accurately characterizing OpenMP application overheads. In *International Workshop on OpenMP*. Springer, 13–25.

[13] Márcio Castro, Luis Fabricio Wanderley Goes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra, and Jean-Francois Mehaut. 2011. A machine learning-based approach for thread mapping on transactional memory applications. In *2011 18th International Conference on High Performance Computing*. IEEE, 1–10.

[14] 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.

[15] 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.

[16] Le Chen, Quazi Ishtiaque Mahmud, Hung Phan, Nesreen Ahmed, and Ali Jannesari. 2023. Learning to parallelize with openMP by augmented heterogeneous AST representation. *Proceedings of Machine Learning and Systems* 5 (2023).

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

[18] 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.

[19] 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.

[20] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. 2023. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062* (2023).

[21] 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).

[22] 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.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[24] 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*. 45–57.

[25] 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.

[26] A. Dutta, J. Choi, and A. Jannesari. 2023. Power Constrained Autotuning using Graph Neural Networks. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 535–545. https://doi.org/10.1109/IPDPS54959.2023.00060

[27] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[28] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. 2013. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of*

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

the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 1–10.

[29] 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.

[30] Akihiro Hayashi, Kazuaki Ishizaki, Gita Koblents, and Vivek Sarkar. 2015. Machine-learning-based performance heuristics for runtime cpu/gpu selection. In *Proceedings of the principles and practices of programming on the Java platform*. 27–36.

[31] Tal Kadosh, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023. Quantifying openmp: Statistical insights into usage and adoption. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[32] 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).

[33] 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.

[34] Jiasen Lu, Dhruv Batra, Devi Parikh, and Stefan Lee. 2019. Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. *Advances in neural information processing systems* 32 (2019).

[35] 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.

[36] Alberto Magni, Christophe Dubach, and Michael FP O'Boyle. 2013. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.

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

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

[39] Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks.

[40] 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.

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

[42] 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.

[43] Isaac Sánchez Barrera, David Black-Schaffer, Marc Casas, Miquel Moretó, Anastasiia Stupnikova, and Mihail Popov. 2020. Modeling and optimizing numa effects and prefetching with machine learning. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–13.

[44] 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.

[45] Nicolai Stawinoga and Tony Field. 2018. Predictable thread coarsening. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 2 (2018), 1–26.

[46] 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).

[47] 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.

[48] 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).

[49] Chen Sun, Austin Myers, Carl Vondrick, Kevin Murphy, and Cordelia Schmid. 2019. Videobert: A joint model for video and language representation learning. In *Proceedings of the IEEE/CVF international conference on computer vision*. 7464–7473.

[50] 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.

[51] Ali TehraniJamsaz, Quazi Ishtiaque Mahmud, Le Chen, Nesreen K Ahmed, and Ali Jannesari. 2024. Perfograph: A numerical aware program graph representation for performance optimization and program analysis. *Advances in Neural Information Processing Systems* 36 (2024).

[52] 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 *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1206–1216.

[53] 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.

[54] Vasily Volkov and James W Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 1–11.

[55] 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.

[56] 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.

[57] Xingfu Wu, Prasanna Balaprakash, Michael Kruse, Jaehoon Koo, Brice Videau, Paul Hovland, Valerie Taylor, Brad Geltz, Siddhartha Jana, and Mary Hall. 2023. ytopt: Autotuning scientific applications for energy efficiency at large scales. *arXiv preprint arXiv:2303.16245* (2023).

[58] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Hal Finkel, Paul Hovland, Valerie Taylor, and Mary Hall. 2022. Autotuning PolyBench benchmarks with LLVM Clang/Polly loop optimization pragmas using Bayesian optimization. *Concurrency and Computation: Practice and Experience* 34, 20 (2022), e6683.

[59] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).