ConTraPh: Contrastive Learning for Parallelization and Performance Optimization

Quazi Ishtiaque Mahmud
Iowa State University
Ames, Iowa, USA
mahmud@iastate.edu

Ali TehraniJamsaz
Iowa State University
Ames, Iowa, USA
tehrani@iastate.edu

Nesreen K. Ahmed
Cisco AI Research
San Francisco, California, USA
nesahmed@cisco.com

Theodore L. Willke
DataStax
Seattle, Washington, USA
ted.willke@datastax.com

Ali Jannesari
Iowa State University
Ames, Iowa, USA
jannesar@iastate.edu

Abstract

With the advancement of HPC platforms, the demand for high-performing applications continues to grow. One effective way to enhance program performance is through parallelization. However, fully leveraging the powerful hardware of HPC platforms poses significant challenges. Even experienced developers must carefully consider factors such as runtime, memory usage, and thread-scheduling overhead. Additionally, achieving successful parallelization often requires running applications to determine the optimal configurations. In this paper, we propose ConTraPh, a framework that integrates Contrastive Learning with Transformers and Graph Neural Networks to capture the inherent parallel characteristics of source programs through a multi-view program representation, utilizing both source code and compiler intermediate representations. This contrastive learning framework allows the model to effectively learn correct parallel configurations from positive samples while avoiding incorrect ones through negative samples. We evaluate ConTraPh on six downstream tasks involving three different parallel programming models OpenMP, OpenCL and, OpenACC that include OpenMP clause prediction, performant reduction style detection, performant scheduling type detection, CPU/GPU parallelism prediction, Heterogeneous Device Mapping for OpenCL code, and OpenACC clause prediction. ConTraPh outperforms state-of-the-art models in these tasks, achieving accuracy improvements of up to 8%, 10%, 7%, 4%, 2%, and 9%, respectively. ConTraPh achieves speedups as high as 13x, 18x, 14x, and 4.4x on the reduction style detention, scheduling type detection, CPU/GPU parallelism prediction, and Heterogeneous Decive Mapping tasks, respectively. Results also demonstrate that ConTraPh can be integrated with third-party tools, such as large language models (LLMs), to enhance the performance of state-of-the-art models like GPT-4 by up to 15% based on established code generation metrics, such as CodeBERTScore. ConTraPh is available at https://github.com/quazirafi/ConTraPh.git.

CCS Concepts

• **Computing methodologies** → **Parallel programming languages**; **Artificial intelligence**; **Knowledge representation and reasoning**; **Probabilistic reasoning**.

Keywords

Contrastive Learning, GNN, LLM, OpenMP, HPC

ACM Reference Format:
Quazi Ishtiaque Mahmud, Ali TehraniJamsaz, Nesreen K. Ahmed, Theodore L. Willke, and Ali Jannesari. 2025. ConTraPh: Contrastive Learning for Parallelization and Performance Optimization. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3721145.3725758

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725758

1 Introduction

Automatic parallelization of source code remains one of the most challenging problems in high-performance computing (HPC). As applications demand increasingly extensive computational resources, the need for frameworks that can automatically optimize program performance has become more critical than ever. This challenge arises from the numerous factors involved, with one of the most significant being the identification of code regions that can be executed in parallel.

However, identifying parallelism opportunities within programs alone is insufficient for achieving significant performance improvements. There are multiple ways and strategies for parallelizing a program, and the optimal approach is often determined through manual inspection of the source code or by employing brute-force methods. This process may involve executing the program in various configurations to identify the best parallelization strategy, which is computationally expensive and can hinder developers' productivity. OpenMP, OpenCL, OpenACC are widely used programming models for writing parallel programs. These programming frameworks offer both CPU and GPU level parallelism opportunities. OpenMP and OpenACC also provide various clauses, such as `private`, `reduction`, `firstprivate`, `lastprivate`, `simd`, `target`, and others, which facilitate parallelization of source programs. Different programs may require specific clauses for efficient parallelization, and it is important to note that not all programs are amenable to parallelization.

A code region may require multiple clauses to ensure proper parallelization, as well as memory and data consistency. Therefore, developers must carefully analyze the program to identify regions suitable for parallelization. Subsequently, they should select the most appropriate subset of clauses to effectively parallelize the identified code region. ConTraPh assists in this process by automatically identifying parallel code regions suitable for parallelization clauses and recommending the appropriate subset of clauses for parallelizing these regions. An essential consideration in parallelization is that simply parallelizing a program is not enough unless it results in a performance gain. One of the primary objectives of program parallelization is to improve runtime efficiency. The same program can be parallelized in various ways, and different parallelization techniques can significantly impact its performance. For instance, as noted by the authors in [23], various OpenMP configurations, such as reduction and scheduling, can influence program performance. However, determining the optimal configuration for a given program is often not straightforward. A careful analysis, frequently accompanied by the execution of the program under different configurations, is essential to make an informed decision. This process is both challenging and computationally expensive. Therefore, ConTraPh aids by automatically identifying optimal configurations for a code region. It generates a ranking of these configurations by analyzing the flow-aware characteristics of the code region from multiple perspectives.

To address this challenging problem, ConTraPh utilizes multi-view program representations to learn underlying features. It employs the source code as a sequence of tokens (a high-level view) and a graph representation of the program, constructed from three types of program flows: control flow, data flow, and call flow, all derived from the compiler's intermediate representation (a low-level view). Additionally, ConTraPh combines the strengths of Transformers and Graph Neural Networks to learn from these views, enabling it to capture features across diverse levels and representations.

In summary, this paper makes the following contributions:

- A novel multi-view learning approach that integrates Transformers and Graph Neural Networks to capture both high-level features from source code and low-level features from compiler intermediate representations.
- The application of the proposed method to identify optimal configurations for parallelization and performance optimization in computational tasks, supporting a wide range of configuration types across CPUs and GPUs.
- Generalization of the method across multiple parallel programming models, including OpenMP, OpenACC, and OpenCL.
- Introduction of two novel downstream tasks, Reduction and Scheduling Style Prediction, in addition to four other downstream tasks to demonstrate the extensibility of the method to diverse parallelization tasks.
- Integration of the proposed method with third-party tools, such as large language models (LLMs), leading to a 15% improvement in GPT-4's parallel code generation as measured by CodeBERTScore.

2 Methodology

In this section, we present the methodology for developing ConTraPh. The entire workflow is illustrated in Figure 1. Six commonly used clauses: `private`, `reduction`, `lastprivate`, `firstprivate`, `simd`, and `target`, along with their combinations are considered in this study, as these are generally effective for parallelizing most loops. For loops that cannot be parallelized, we assign the label `'none'`, indicating they are not suitable for parallel execution.

2.1 Dataset Preprocessing

For pre-training, we utilize the OMP_Serial dataset [8] to some extent. However, since this dataset only contains `private` and `reduction` clauses, we augmented it with additional parallel configurations, such as `lastprivate`, `firstprivate`, `simd`, `target`, and combinations of these clauses. To enhance the dataset, we extracted code containing the mentioned clauses from well-established GitHub repositories and HPC benchmarks. To ensure the quality of the dataset, we employed GitRank [15] to rank the repositories, considering only those with an overall score above 50.0. Each source code is included in our dataset only if it is compilable, as we use both LLVM IR and source code representations for

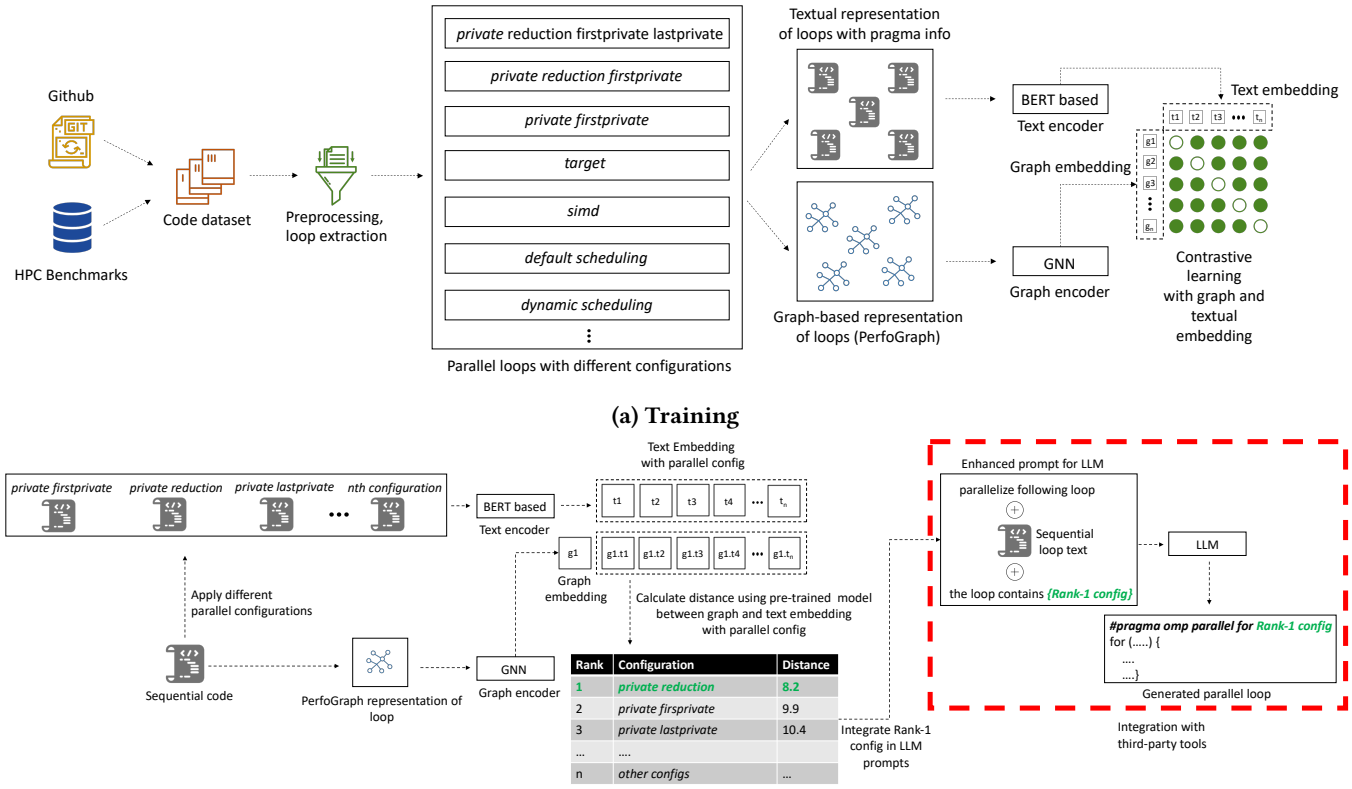

Figure 1: Overview of the ConTraPh workflow.

ConTraPh, and obtaining LLVM IR requires successful compilation.

To improve the dataset's quality, each source code is processed using two parallelism assistant tools: AutoPar [27] (Static Analyzer) and DiscoPoP [20] (Dynamic Analyzer), to verify the correctness of the parallel configurations. The source code that passes these filters is then prepared for pre-training. Specifically, loops deemed suitable for parallelization, along with their associated clauses, are extracted for contrastive pre-training. Lastly, the LLVM Intermediate Representations (IRs) of the extracted loops are obtained.

Finally, a set of transformation flags [3] from LLVM is applied to the IRs to enhance the dataset, following the methodology outlined in [36]. For evaluation, we use source codes from the AutoParBench suite [25], EPCC Benchmark [17], and PolyBench-OpenACC Benchmark [12]. To ensure the integrity of our test set, we carefully excluded any code from these repositories during the training phase. The resulting dataset comprises approximately 700 source files, from which 1,517 loops were extracted. After applying the transformation flags, we generated roughly 30,000 IR files to pre-train ConTraPh. Of these, 10% were reserved for validating the contrastive pre-training pipeline.

2.2 Generating Program Representation and Embedding

ConTraPh leverages both textual and graph-based representations of programs, as illustrated in Figure 2. The textual representation enables the model to capture the syntactical and semantic characteristics of source programs. Meanwhile, the graph-based representation explicitly exposes various flows, such as control, data, and call flows. Capturing these dependencies is extremely beneficial for program optimization and parallelization, as demonstrated in the results section.

2.2.1 Textual Representation and Text Encoder. As outlined in the section 2.1, the textual representation includes the loop along with its parallel configurations. For encoding, we utilize the CodeBERT model [41] to generate embedding vectors. The default CodeBERT configuration in PyTorch is employed, with a maximum input length of 512 tokens. Text inputs shorter than this length are padded to complete the sequence. To enhance the model's understanding, token

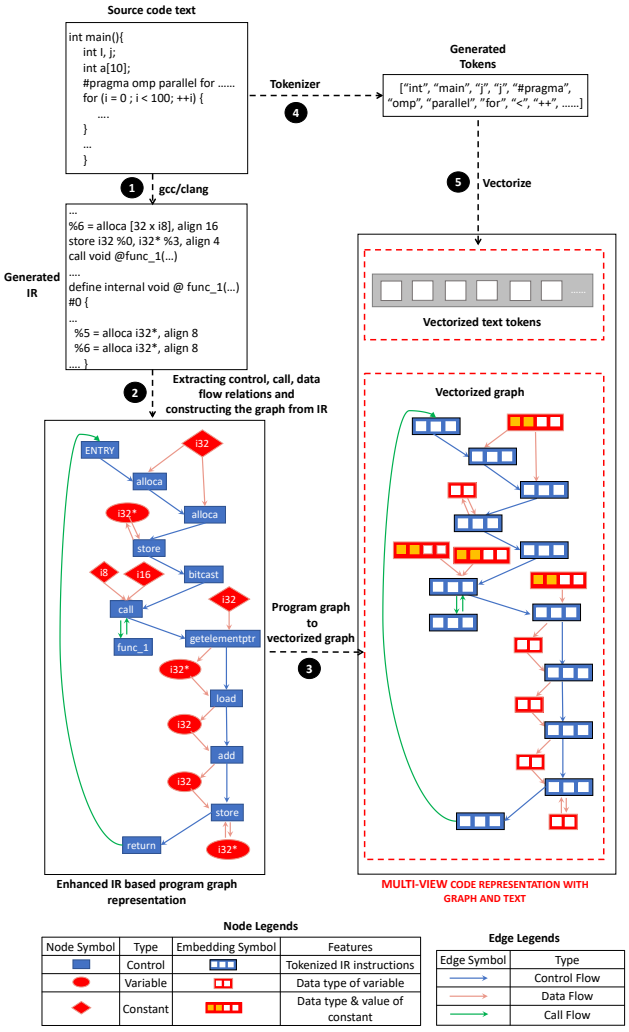

Figure 2: Muti-view code representation with graph and text.

weights are assigned, emphasizing specific tokens. For each loop, higher weights are allocated to tokens that represents the parallel constructs (e.g., 'private', 'reduction', 'simd' etc.) to highlight the parallel configurations. The final text embedding is derived by combining the weighted tokens with the embeddings produced by CodeBERT.

2.2.2 Graph Representation and Graph Encoder. For each loop, we have the corresponding IR. To derive a graph representation from the IR, we utilize the state-of-the-art IR-based program representation, PerfoGraph[35]. This representation integrates control flow, call flow, and data flow. PerfoGraph has proven effective in prior applications such as parallelism detection and parallel pattern recognition [35], making it an ideal choice for modeling parallel programs.

PerfoGraph contains mainly three types of nodes: Control, Variable, and Constant. Control nodes represent the instructions in IR statements and Variable and Constant nodes represent the variables and constants associated with those instructions in IR statements. PerfoGraph uses three types of edges: Control flow edges, Call flow edges, and Data flow edges to model the relationship across different nodes in the program graph. For encoding and generating embedding vectors for the graph representation of programs, we use Heterogeneous Graph Neural Networks (HGNNs). Heterogeneous Graph Neural Networks (HGNNs) can support modeling different types of nodes and edges, and PerfoGraph also has multiple types of nodes and edges. A Gated Attention Network (GAT) [38] is employed for modeling the relationships among different elements of the program graph with 3 GATConv layers based on PyTorch. GATs are helpful as they allow for assigning different weights to different nodes in a neighborhood. This suits our problem since we also assign different weights to different tokens in the program graph.

To generate embedding for the PerfoGraph representation of a loop, first, the embedding of each node in the PerfoGraph representation is obtained using PyTorch learnable embeddings. A pooling is applied to combine the embeddings of each node type. For the pooling operator, we use the PyTorch-based global_mean_pool function. After obtaining the embeddings for each node type we concatenate them together to obtain the final embedding of the entire graph.

2.2.3 Applying Projection. After embedding text and graph, there is a projection step. Contrastive training and loss require the embedding of two views to be the same. The projection step ensures this requirement. In fact, projection passes the embeddings of both text and graph through a linear transformation layer to obtain the same-sized embedding vectors for both views. The projection vector size is set to 128 in this study, meaning the length of text embedding and graph embedding would be 128 for a particular loop.

2.3 Contrastive Pre-training

2.3.1 Designing the Positive and Negative Samples. Contrastive learning is particularly suited for tasks that involve clustering similar samples (positive samples) while separating dissimilar ones (negative samples). This capability enables the model to align text and graph representations effectively, allowing it to learn from both. A crucial step in applying contrastive learning is constructing the positive and negative samples for each data point, where each data point represents a single loop. Each loop can be parallelized using various parallel constructs, though not all configurations are optimal or correct. The positive and negative samples are, therefore, designed to guide the model in identifying the

most suitable configurations for parallelizing a given loop while avoiding suboptimal or incorrect ones.

- Positive Samples: The parallel constructs the developer originally added are considered as the positive samples.
- Negative Samples: The negative samples represent all incorrect configurations for a particular loop. The negative samples are found by generating all combinations of six clauses and then excluding the correct configuration from the set.

For example, if the correct OpenMP configuration for a particular loop is `#pragma omp parallel for private(var) reduction(op: var)`, then the positive sample will contain the correct OpenMP configuration, and all other configurations like `#pragma omp parallel for private(var)`, `#pragma omp parallel for private(var) lastprivate(var)` etc. are considered as the negative samples.

2.3.2 Designing Contrastive Loss Function. The goal of contrastive pre-training is to enable the model to generate similar vector representations for a loop with a correct parallel configuration and the IR of that loop. When this occurs, the Euclidean distance between the two representations will be small, indicating that a potential correct parallel configuration has been found for that loop. We use the projected graph and text embeddings for calculating the contrastive loss. As we employ a distance-based approach for measuring the contrastive loss, instead of using the standard contrastive learning objective, we employ a modified version, which is known as mean-shifted contrastive (MSC) loss function as described in [28]. It maximizes the distance between positive and negative samples, hence, helps ConTraPh to learn which configurations are better (positive samples) for a particular loop and which need to be avoided (negative samples). First, we have the projected and normalized graph and text embedding vectors txt_v and grp_v of size (B, D) where B is the batch size and D represents the number of elements in each vector. We concatenate these two vectors and construct out where,

$$out = txt_v \oplus grp_v$$

Here out is of shape $(2B, D)$. Then, we compute the similarity matrix:

$$sim\_matrix_{ij} = \exp\left(\frac{\sum_{k=1}^{D} out_{ik} \odot out_{jk}}{temp}\right)$$

Here $\odot$ represents dot product between vectors and $i, j = 1, 2, 3, \ldots, 2B$. Hence, sim_matrix_{ij} is of size $(2B, 2B)$. sim_matrix_{ij} computes a full similarity matrix that considers all possible pairs of vectors, including both text-text vectors (within group pairs) and graph-text vectors (cross-group pairs). We calculate the positive similarity between corresponding vectors txt_v and grp_v using the following:

$$pos\_sim_i = \exp\left(\frac{\sum_{j=1}^{D} txt_{v_{ij}} \odot grp_{v_{ij}}}{temp}\right)$$

Here $i = 1, 2, 3, \ldots, B$. The output pos_sim is a $1D$ vector of size B where each element corresponds to the similarity score of a pair of corresponding vectors from txt_v and grp_v. Then, we concatenate the pos_sim with itself, effectively doubling its size to $2B$. This concatenation ensures that the positive similarities align with their corresponding entries in the sim_matrix. This helps in the calculation of the loss function as it compares the positive similarity scores ('pos_sim') against all other similarities in the 'sim_matrix'. Finally, the loss is calculated as the mean of the negative log of the ratio between the positive similarity and the sum of all other similarities using the following formula, encouraging the model to make the positive pairs more similar relative to other pairs in the batch.

$$loss = -\frac{1}{2B}\sum_{i=1}^{2B} \log\left(\frac{pos\_sim_i}{\sum_{j=1}^{2B} sim\_matrix_{ij}}\right)$$

We adapt the PyTorch implementation of MSC loss in ConTraPh. The Adam optimizer is used for the training and the learning rate is set to 0.001. We set the batch size to 100, and the dimension of each vector is chosen as $D = 120$. We empirically choose a relatively low value of $temp = 0.25$ to sharpen the contrast between positive and negative pairs, making the model more effective at learning distinct representations. The model is trained for 600 epochs, and the checkpoint with the least loss is saved for later use. The total training time for the model is two days, 4 hours, 47 minutes, and 36 seconds. Figure 3 shows the epoch vs. loss curve for the pre-training process.

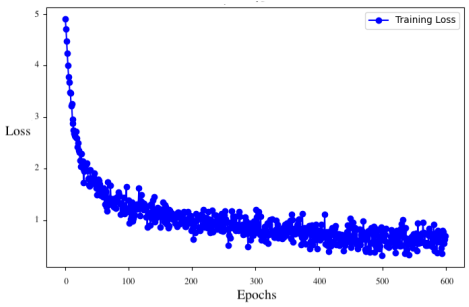

Figure 3: Epoch vs loss for Contrastive Pre-training

2.4 Validating the Pre-trained Model

We first generate the text embedding vector with all parallel configurations using our pre-trained model to predict parallel configurations for a particular loop. Then, the graph embedding vector is generated using the IR-based PERFOGRAPH representation of that loop using the same model. We calculate the Euclidean distance between the generated text embedding vectors and the graph embedding vector (Figure 1.b). Then, a ranking based on the distance of the parallel configurations for that particular loop is generated.

The ranking is created in an increasing order meaning the parallel configuration that is at least distance from the loop is at the top of the ranking. We choose the parallel configuration that is at the top of the ranking as this configuration represents the most optimal configuration for the loop. Table 1 represents the results for the validation set, which has around 3k IR files. The individual accuracies of the parallel clauses are obtained by checking if the top-ranked parallel configuration matches exactly with the ground-truth values obtained from developers.

Table 1: Parallel clause detection accuracy of CONTRAPH on the validation set

Clause	Detection Accuracy
private	0.9879
reduction	0.9388
firstprivate	0.9412
lastprivate	1.0000
simd	0.9313
target	0.9224
none	1.0000
Overall Accuracy	**0.9602**

3 Results and Experiments

CONTRAPH is evaluated against 6 downstream tasks related to performance optimization and parallelization. In addition, we integrate CONTRAPH with third-party LLMs and demonstrate that it can also help enhance the performance of these models. In this section, we discuss the results of the downstream tasks. For all experiments, we used computing nodes with the same configurations: two NVIDIA Tesla V100-32GB GPUs and two 18-core Intel Skylake 6140 CPUs.

3.1 Predicting OpenMP Configuration for CPU-based Parallelism

For this task, CONTRAPH is responsible for detecting the OpenMP configuration given a particular loop. This task is focused on detecting CPU-based parallelism opportunities, so only the OpenMP clauses `private`, `reduction`, `firstprivate`, `lastprivate`, and a combination of them are considered. The pre-trained CONTRAPH model is used for experiments.

The test set is created from the AutoParBench repository. It contains three benchmark applications: NAS Parallel Benchmark (NPB) [16], Rodinia [6], and Dataracebench [21] along with the OpenMP configurations from expert developers, which are considered as ground-truth values. Therefore, the results from CONTRAPH can be compared to these ground-truth values. NPB has eight applications, and 90 loops (BT: 7, IS: 6, CG: 10, FT: 5, EP: 6, LU: 13, MG: 15, and SP: 28) are extracted from these applications. From 4 Rodinia Benchmark applications total of 21 loops (BFS: 1, B+ Tree: 6, Heartwall: 13 and 3D: 1) are extracted. Then, 34 loops are extracted from the DataRaceBench benchmark. So, around 145 loops are extracted in total. Each loop can have multiple OpenMP clauses. We show the details of the different types of clauses in Table 2.

Table 2: Different OpenMP clauses extracted from AutoParBench

OpenMP Clause	Number samples in test set
private	117
reduction	61
firstprivate	46
lastprivate	14
none (non-parallel)	12
Total clauses	**250**

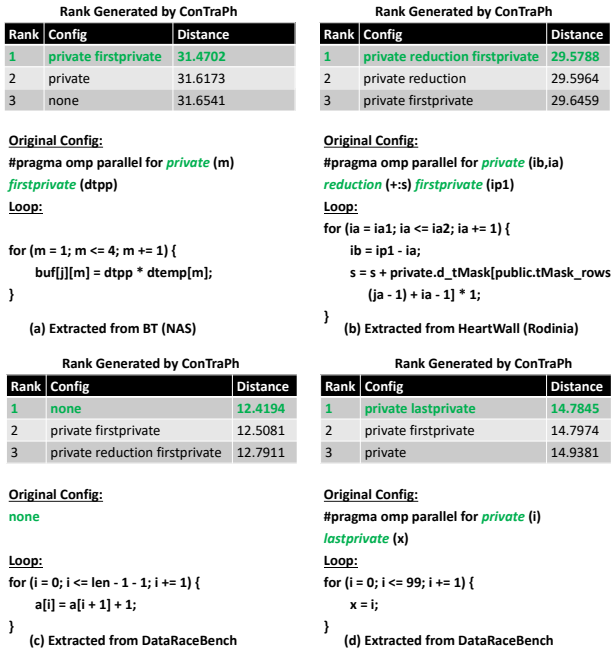

Figure 4: Illustration of the output of CONTRAPH. For each loop, the output configuration is shown up to 3 ranks. For generating the results in Table 3 only the Rank-1 configurations are used

Table 3: Comparing OpenMP clause detection accuracy across different models

Model	OpenMP Clause Detection Accuracy					Overall accuracy
	private	reduction	firstprivate	lastprivate	none (Non-parallel)	
Pragformer	0.7312	0.7219	N/A	N/A	0.5833	0.6788
Graph2Par	0.7944	0.7839	N/A	N/A	0.6667	0.7483
PrograML	0.8765	0.8329	N/A	N/A	0.75	0.8198
PerfoGraph	0.9001	0.8776	N/A	N/A	0.75	0.8426
AutoParLLM	0.9004	0.8785	N/A	N/A	0.8333	0.8707
ConTraPh	**0.9829**	**0.9344**	**0.9565**	**1.0**	**0.9167**	**0.9581**

Table 4: Comparing performance of ConTraPh with other approaches on Reduction Style prediction task. OS = 'Oracle Speedup'.

Model	Reduction Styles									Overall accuracy	Speeedup (OS = 17.49X)
	precision			recall			f1-score				
	Atomic	Clause	Critical	Atomic	Clause	Critical	Atomic	Clause	Critical		
PrograML	0.50	1.0	0.83	0.83	0.57	0.71	0.62	0.73	0.77	0.70	3.25x
PerfoGraph	0.60	0.88	0.71	0.50	0.78	1.0	0.55	0.82	0.83	0.75	7.62x
AutoParLLM	0.67	0.67	1.0	0.89	0.67	0.62	0.76	0.67	0.77	0.75	8.21x
ConTraPh	**0.80**	**0.67**	**1.0**	**0.89**	**0.67**	**0.88**	**0.84**	**0.67**	**0.93**	**0.85**	**13.51x**

For generating the results with ConTraPh, we first extract all the loops and their associated OpenMP clauses. Since ConTraPh is designed to predict OpenMP configurations, we remove the existing configurations from the loops. Next, we generate the IR representation for each loop. From this IR, we also produce the PerfoGraph representation. The input to ConTraPh consists of the sequential loop without the OpenMP clause and its corresponding program graph (PerfoGraph) representation. Based on these inputs, ConTraPh ranks the OpenMP configurations according to distance, as described in section 2.4. The top-ranked OpenMP configuration is then selected as ConTraPh's prediction. Table 3 presents the results. We obtained open-source model checkpoints for PragFormer, Graph2Par, ProGraML, PerfoGraph, and AutoParLLM for comparison and applied these pre-trained models to our test set of 250 loops. However, none of these models account for `firstprivate` and `lastprivate` OpenMP clauses, so they are marked as N/A in the table.

As shown in Table 3, ConTraPh achieves up to 7% better accuracy than the state-of-the-art models. Additionally, Figure 4 illustrates how rankings are generated for the different OpenMP configurations. Although we display up to the Rank-3 configurations in Figure 4, ConTraPh only selects the top-ranked configuration as the final output. Figure 4 illustrates the capability of ConTraPh to generate correct OpenMP configurations involving multiple OpenMP clauses (Figure 4(a, b, d)). It illustrates the capability of ConTraPh to detect non-parallel loops (Figure 4(c)). It is observable from Figure 4 that all Rank-1 configurations predicted by ConTraPh match with the developer's version (Original Config) of the OpenMP configuration.

3.2 Predicting Performant Coding Styles (Reduction)

For the previous task, ConTraPh was evaluated on the OpenMP clause prediction task. However, a single clause can be applied in multiple ways, potentially affecting the performance of programs differently, as shown in the study by [23]. Therefore, in this downstream task, we evaluate ConTraPh on detecting the most performant coding styles that involve the OpenMP clause `reduction`. Specifically, We consider three ways to implement a parallel loop with `reduction` clause: i) `Atomic`, ii) `critical` and iii) `clause` based reduction as discussed by authors of [23].

For evaluating ConTraPh on detecting performant reduction styles, we use the dataset from [23]. This dataset contains 60 source files, each with the throughput of all three reduction styles. By applying LLVM transformation flags, we increase the sample size to 200 (85 `atomic`, 62 `clause`, 53 `critical`). The programs include graph algorithms like BFS, Single Source Shortest Path, Connected Components, etc. Throughputs are calculated using Equation 1, which measures the number of edges processed per microsecond.

$$\text{Throughput} = \frac{\text{\# of edges processed}}{\text{runtime in seconds} \times 1000000.0} \quad (1)$$

For a particular program with a given input, the number of processed edges will be the same. From Equation 1, we observe that throughput is inversely proportional to runtime. Therefore, higher throughput corresponds to reduced runtime, indicating a speedup in the program. As a result, it is reasonable to assume that increasing throughput is equivalent to achieving speedup. We use this throughput to create a labeled dataset for the three reduction styles and apply Equation 1 to calculate the speedup gain in our results. For

Table 5: Comparing performance of ConTraPh with other approaches on Scheduling Style prediction task. OS = 'Oracle Speedup'.

Model	Scheduling Styles						Overall accuracy	Speeedup (OS = 21.25X)
	precision		recall		f1-score			
	static	dynamic	static	dynamic	static	dynamic		
PrograML	0.67	0.87	0.86	0.67	0.75	0.76	0.76	4.19x
PerfoGraph	0.78	0.81	0.84	0.73	0.81	0.77	0.79	6.21x
AutoParLLM	0.82	0.84	0.87	0.78	0.84	0.81	0.83	11.02x
ConTraPh	**0.88**	**0.92**	**0.93**	**0.85**	**0.90**	**0.89**	**0.90**	**18.96x**

comparing ConTraPh with other models regarding speedup, we use the concept of **'Oracle Speedup'**. **'Oracle Speedup'** is obtained when a model always selects the best runtime for a program. For each program, the best-performing reduction style (i.e., the reduction configuration with the highest throughput) is assigned as the label for that program. The task then becomes predicting which of the three reduction styles will provide the best throughput for a given program using ConTraPh. Note that ConTraPh has already been pre-trained on different OpenMP clauses. In this phase, we fine-tune ConTraPh using this reduction style dataset.

We keep 10% (8 `atomic`, 6 `clause`, 6 `critical`) of the dataset for testing, with the remaining data used for fine-tuning. The goal of the fine-tuning remains the same: to enable the model to generate similar vector representations given a loop with a correct OpenMP configuration and the IR of that loop. This will result in less Euclidean distance between vectors, indicating that a possible correct OpenMP configuration is found for that loop. The only difference here is that, instead of using all OpenMP configurations, we focus exclusively on the three reduction styles as our target OpenMP configurations. The inference phase remains identical. For predicting the best reduction style for a particular loop, we first generate the text embedding vectors for all reduction styles using the fine-tuned ConTraPh model. Then, the graph embedding vector is generated using the IR-based PerfoGraph representation of that loop, also with the fine-tuned model. The generated text embedding vectors are then compared with the graph embedding vector to calculate the Euclidean distance. A ranking is generated for the reduction styles where the style with the least distance is at the top, and the top-ranked reduction configuration is selected as the output of ConTraPh. To the best of our knowledge, we are the first to apply a Contrastive Learning method to predict the most performant reduction style for a particular loop. No previous work has attempted to solve this problem in an autonomous manner, making it difficult to compare ConTraPh's performance directly with other methodologies. However, because the detailed implementations of PrograML, PerfoGraph, and AutoParLLM were available, we were able to compare ConTraPh with these three approaches. Table 4 shows the results.

As shown in the Table 4, ConTraPh achieves up to 10% better accuracy than the current best Deep Learning approaches for parallelization discovery and performance optimization tasks, such as PrograML and PerfoGraph. PrograML uses a GGNN as its core model architecture, whereas both PerfoGraph and AutoParLLM use RGCN as its core model with a Cross-entropy-based loss function. All three architectures use only LLVM IR-based representation of source programs. In contrast, ConTraPh leverages both LLVM IR-based graph and textual representation of source code, employs MSC-based contrastive loss, and uses Gated Attention Networks (GATs) in its model architecture. ConTraPh also achieves the highest speedup (13.51X) than all other SOTA models, and it is very close to the **'Oracle Speedup'** of 17.49X.

3.3 Predicting Performant Coding Styles (Scheduling)

Like the `reduction` clause, the `schedule` clause in OpenMP also has different styles or configurations. In OpenMP, a loop can be parallelized using the `static` scheduling, which assigns a certain chunk of loop iterations to each of the threads statically. Alternatively, `dynamic` scheduling is available, where chunks of iterations are assigned to threads at runtime. These different scheduling configurations can significantly impact program performance.

In this downstream task, we evaluate ConTraPh on detecting the most performant coding styles involving the OpenMP clause `schedule`. Specifically, we focus on two types of scheduling: i)`static` and ii)`dynamic`. To evaluate ConTraPh, we utilize the dataset from [23], which contains 167 source files, each annotated with runtime information for both `static` and `dynamic` scheduling. Similar to the previous task, we use this runtime information to create a labeled dataset for the two scheduling types. Additionally, LLVM transformation flags are applied to increase the dataset size to 855 samples (430 `static`, 425 `dynamic`). The task is to predict, given a program, whether `static` or `dynamic` scheduling will provide the best runtime. We further fine-tune ConTraPh using this scheduling style dataset; 10% (45 static, 41 dynamic) of the dataset is reserved for testing, and the remainder is used for fine-tuning. The goal of the fine-tuning

remains the same as the reduction style detection model, with the only difference being that instead of detecting three reduction styles, we now target the two scheduling styles as our OpenMP configuration.

During inference, to predict the best scheduling options for a particular loop, we first generate text embedding vector for both scheduling styles using the fine-tuned ConTraPh model. Then, the graph embedding vector is generated using the IR-based PerfoGraph representation of that loop with the same fine-tuned model. Then we use the same methodology as described earlier to create a ranking for the different scheduling styles for a particular loop based on the euclidean distance among the generated vectors. To the best of our knowledge, this is the first work that addresses automatic scheduling style prediction using a Deep Learning approach. However, having access to the training models of ProgramL, PerfoGraph, and AutoParLLM allowed us to compare ConTraPh with these three approaches. Table 5 shows the results.

We observe that ConTraPh achieves up to 7% higher accuracy compared to the current best Deep Learning approaches, such as ProgramL and PerfoGraph, which have been previously applied to parallelization and performance optimization tasks, including parallelism discovery, parallel pattern detection, and device mapping. Also, ConTraPh achieves better speedup (18.96X) than other SOTA models, and the achieved speedup is also very close to the **'Oracle Speedup'** of 21.25X.

3.4 CPU vs GPU Parallelism

OpenMP allows for CPU-based parallelism using `simd` clause and GPU-based parallelism using the `target` clause. Some programs may benefit more by CPU-based parallelism using loop vectorization, while others, particularly those involving data-intensive computations, may benefit more from GPU offloading. However, like other OpenMP configurations, determining the most suitable configuration often requires careful inspection of source programs and program execution, which can be daunting and computationally expensive, especially for programs with long execution times.

In this task, we use ConTraPh to automatically distinguish between source programs that are better suited for parallelization using CPU and GPU. Given a loop, the goal is to predict whether that program should be parallelized using OpenMP `simd` or `target` directive. As there is currently no such dataset available, we turn to the open-source repositories and benchmarks, only considering repositories with an overall_score over 50.0, calculated using GitRank, as mentioned earlier. Additionally, we only consider programs that are both compileable and executable. Since OpenMP annotations in open-source repositories can often contain noisy samples, we executed each program five times and collected the execution times for both `simd` and `target` configurations in OpenMP. We labeled a program as a `simd` program if its average runtime using the `simd` directive was less than its average runtime using the `target` clause. Similarly, we labeled loops that performed better with the target directive. After preprocessing and applying LLVM IR-based transformations, we obtain 1,612 (1,207 simd, 405 target) loops. We use around 10% (112 `simd`, 47 `target`) of the data for testing ConTraPh. For inference, as described earlier, the text embeddings for the two configurations are generated using ConTraPh and compared with the graph embedding of a particular code and ranking is generated based on the euclidean distance.

For this task, we compared ConTraPh with recent works such as ProgramL, PerfoGraph, and AutoParLLM, as these models address similar problems like CPU/GPU device mapping and parallelization pattern detection. The results are presented in Table 6. We observe that PerfoGraph and AutoParLLM have very similar performances likely due to their similar GNN architectures and program representations. AutoParLLM, in fact, uses PerfoGraph to model the flows in programs, which is one of the core components of its architecture. Both PerfoGraph and AutoParLLM have a slight advantage over ProgramL, which also uses the IR representation as the other two models. However, ProgramL does not have performance-specific optimizations in the IR-based program graphs like PerfoGraph and AutoParLLM. We believe that this is the main reason for ProgramL achieving lower accuracies in most downstream tasks than PerfoGraph and AutoParLLM.

However, ConTraPh achieves better results than these models. In this task as well, ConTraPh demonstrates 4% higher accuracy in predicting the appropriate use of `simd` and `target` clauses compared to the best-performing models. Also, ConTraPh achieves the best speedup (14.58X) among other models, and the achieved speedup is extremely close to the **'Oracle Speedup'** of 16.15X.

3.5 Complete OpenMP Clause Generation

Here, we demonstrate how ConTraPh can be integrated with third-party tools like LLMs to enhance autonomous parallelization and program optimization. To showcase the usefulness of ConTraPh, we selected the OpenMP clause generation task. In this take, given a loop, the objective is to automatically generate the complete OpenMP clauses required to parallelize the loop. ConTraPh is capable of generating suitable OpenMP clauses such as `private`, `reduction`, `firstprivate`, `lastprivate`, or combinations of these clauses. However, ConTraPh does not generate the variables and operators associated with these clauses. `private`, `firstprivate`, `lastprivate` clause contains variables list associated with

Table 6: Comparing performance of ConTraPh with other approaches on CPU and GPU based parallelization. OS = 'Oracle Speedup'

Model	CPU/GPU configurations						Overall accuracy	Speeedup (OS = 16.15X)
	precision		recall		f1-score			
	simd	target	simd	target	simd	target		
ProgramL	0.90	1.00	1.00	0.58	0.95	0.73	0.91	6.67x
PerfoGraph	0.91	1.00	1.00	0.70	0.95	0.82	0.92	9.82x
AutoParLLM	0.91	1.00	1.00	0.69	0.95	0.82	0.92	10.09x
ConTraPh	**0.95**	**1.00**	**1.00**	**0.82**	**0.98**	**0.90**	**0.96**	**14.58x**

" Parallellize the following loop with OpenMP
{*Loop text here*} "
(a)

" Parallellize the following loop with OpenMP using clauses
{***OpenMP clauses predicted from ConTraPh here***}
{*Loop text here*} "
(b)

Figure 5: (a) Shows the prompts without integrating the results from ConTraPh. (b) Shows the prompts after integrating the results from ConTraPh.

them (e.g., `private(i, j)`), and `reduction` clause is associated with both variables and reduction operators (e.g., `reduction(+:sum)`). In this task, we employ LLMs to automatically generate these elements of OpenMP clauses using the predictions from ConTraPh. Specifically, we extracted 60 loops containing OpenMP clauses from AutoParBench, which includes three benchmarks: NPB, Dataracebench, and Rodinia. For each loop, we generated the OpenMP configurations using ConTraPh following the methodology discussed in section 3.1. These predictions are then incorporated into the prompt for the LLMs. The prompt instructs the LLM to parallelize the loop using the OpenMP clauses predicted by ConTraPh. The input to the LLM consists of the loop text and the OpenMP clauses predicted by ConTraPh. We found that integrating ConTraPh predictions with LLMs significantly enhanced the LLMs' ability to generate complete OpenMP clauses. To illustrate the impact of ConTraPh on LLMs, we design two types of prompts: i) without ConTraPh predictions (Figure 5.a) and ii) with ConTraPh predictions (Figure 5.b). LLMs are invoked with both types of prompts. From the generated responses, we compare the generated OpenMP clauses with the ground-truth values from benchmarks. To evaluate the quality of the generated OpenMP clauses, we used well-known text-generation metrics such as BLEU [26] and ROUGE-L [22], and also code-specific evaluation metrics, including CodeBLEU [29] and CodeBERTScore [41]. The results are reported in Table 7. We can observe that, ConTraPh helps to improve the performance of the most powerful LLM to date, GPT-4, by 15% in terms of CodeBERTScore. The contrastive learning-based prompting helps the LLMs to have some background information before generating the OpenMP clauses. It helps in its reasoning, which leads to better results. Figure 6 shows the effects of ConTraPh on GPT4 and Gemini-Pro output.

Table 7: Comparing OpenMP clause generation results using third-party LLMs. (A higher score indicates a closer match with ground truth and a score of 100 indicates a perfect match)

Model	BLEU	Rogue-L	CodeBLEU	CodeBERTScore
GPT-3.5 [1]	34.02	52.85	48.96	79.0
ConTraPh-GPT-3.5	**63.91**	**92.53**	**68.25**	**96.2**
GPT-4 [2]	39.29	58.68	53.31	81.0
ConTraPh-GPT-4	**67.83**	**94.12**	**70.21**	**96.8**
CodeLlama-34b [30]	30.00	52.00	50.48	77.6
ConTraPh-CodeLlama-34b	**43.94**	**81.60**	**58.47**	**90.9**
Gemini [34]	41.37	56.08	54.63	80.6
ConTraPh-Gemini	**50.62**	**87.30**	**67.48**	**94.2**

3.6 OpenACC Configuration Prediction

We also apply ConTraPh for detecting the proper parallel configurations of another widely used parallel programming framework: OpenACC. One of the reasons for choosing OpenACC is because it offers almost all configurations that ConTraPh is pre-trained with, and also, there are benchmarks with ground truth values which we can compare our results with. We considered two OpenACC benchmarks: EPCC Benchmark [17] and PolyBench-OpenACC Benchmark [12]. Total 78 loops are extracted from the two benchmarks. We found 4 different parallel configurations in these 78 extracted loops: `private`: 31, `firstprivate`: 20, `reduction`: 15, and 12 loops that are not parallelizable. Loops that can not be parallelized are labeled as `none`. After extracting the loops, we generate the IRs and then create the PerfoGraph representation of the loops. Then, we use the contrastively pre-trained ConTraPh model (as introduced in Section 2.3 and 2.4) on this dataset of 78 loops. The pre-trained ConTraPh takes as input both the loop text and the PerfoGraph representation to generate the configurations. Then, we incorporate the ConTraPh-generated configurations into GPT-4 prompts

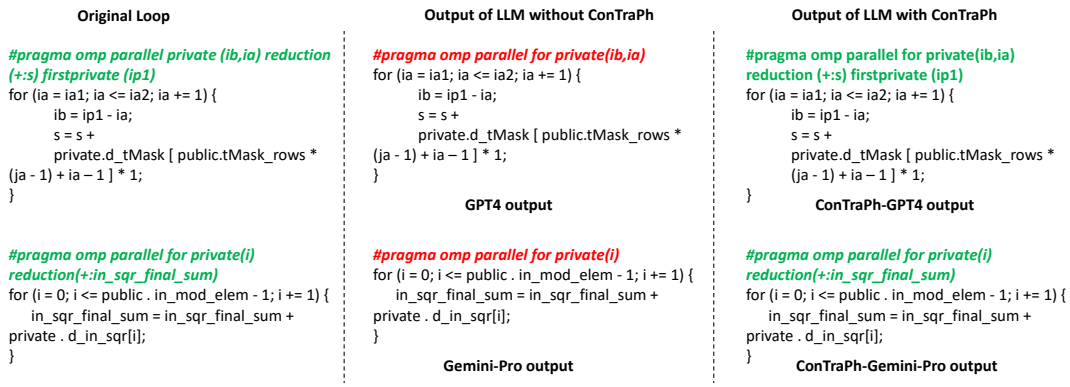

Figure 6: The impact of ConTraPh on LLMs. The outputs of LLMs for OpenMP clauses improve significantly after the integration of ConTraPh into LLMs.

and instruct GPT-4 to generate the complete OpenACC configurations instead of OpenMP using the same procedure as discussed in Section 3.5. As we are able to obtain the pre-trained model checkpoints for ProgramL, PerfoGraph, and AutoParLLM; we compare with these SOTA approaches. however, as none of the models support `firstprivate`, we list them as N/A. From the results of Table 8, it can be observed that ConTraPh achieves better results than all other SOTA models, and it was able to detect all `reduction` and 'non-parallel' loops correctly. We also investigated the mistakes made by ConTraPh and found that there are mismatches regarding the `private` and `firstprivate` clauses for the loop '`for(i=0; i<n; i++) { arr[i] = y[i]; }`'. ConTraPh correctly suggested the `private` clause on variable `i` and `firstprivate` on the array variable `y`. However, the loop counter `i` is by default `private`, and often developers omit explicitly adding `private` clause on such cases. In this case, as the developer did not add this in the benchmark it is considered as a mismatch in our evaluation. And instead of choosing `firstprivate` the developer choose the `copyin` clause on variable `y` to make it shared among all the threads in the accelerator device. As there is no write operation on variable `y`, both configurations are correct and will not affect the correctness of the program.

Table 8: Comparing performance of ConTraPh with state-of-the-art models for OpenACC configuration prediction

Model	OpenACC Clause Detection Accuracy				Overall Accuracy
	private	firstprivate	reduction	none	
ProGraML	77.42	N/A	73.33	75.00	75.64
PerfoGraph	83.87	N/A	86.67	83.33	84.62
AutoParLLM	87.09	N/A	86.67	91.67	88.48
ConTraPh	**96.77**	**95.00**	**1.00**	**1.00**	**97.44**

3.7 Heterogeneous Device Mapping for OpenCL Code

Previous tasks primarily focus on parallelization and optimizing code performance using different parallelization styles. To assess how our approach performs on a different task, we apply ConTraPh to the problem of predicting the appropriate hardware configuration for a specific piece of code. For this task, we use the dataset published in [10]. It contains 681 LLVM IRs extracted from 256 OpenCL kernels. Every datapoint contains the runtime information on one CPU (Intel Core i7-3820, with 3.6 GHz frequency and 8GB memory) and two GPUs (AMD Tahiti 7970, with 1000 MHz frequency, 3GB memory, and NVIDIA GTX 970, with 1050 MHz frequency, 4GB memory). Based on these three hardware configurations, two datasets are created: one considering the CPU and AMD GPU and one considering the CPU and NVIDIA GPU. 276 kernels show better performance in the AMD GPU, while 395 kernels show better performance in the CPU. For NVIDIA GPU, 385 kernels have better runtimes with GPU, and 286 kernels have better runtimes with CPU. For each of the datasets, the task for ConTraPh is: given an OpenCL kernel, predict whether the kernel performs better on a CPU or GPU. Here, we also use the pre-trained ConTraPh model to generate the text and graph embedding vectors, and the same distance-based ranking methodology is used to select the best configuration. To compare with other studies, we follow the same 10-fold validation in the fine-tuning phase. At each step, we maintain a conventional data split: 80% for fine-tuning, 10% for validation, and 10% for testing. At each fold, the 10% test split is chosen uniquely so that a complete prediction can be constructed over the whole dataset. The results in Table 9 show that ConTraPh performs better than state-of-the-art models in most cases. Specifically, ConTraPh achieves state-of-the-art 95.52% accuracy in predicting the optimal device for program execution on the

AMD dataset and near state-of-the-art 92.51% accuracy on the NVIDIA dataset.

Following the studies of Grewe et al. [13], DeepTune [10] and inst2vec [4], the baseline for speedup is considered to be static mapping. Static mapping selects the device that has the best average case performance across all programs in the datasets. For the AMD dataset selecting the CPU yields the best average performance and for the NVIDIA dataset selecting the GPU results in the best average case performance. The numbers reported in Table 9 represent speedups over the static mapping baseline. It can be observed that ConTraPh provides better speedup compared to all state-of-the-art models.

Table 9: Comparing performance of ConTraPh with state-of-the-art models for the task of Device Mapping

Model	Accuracy(%)		Speedup(X)	
	AMD	NVIDIA	AMD	NVIDIA
Grewe et al.	73.38	72.94	2.91	1.26
DeepTune	83.68	80.29	3.34	1.41
inst2vec	88.09	86.62	3.47	1.44
ProGraML	86.6	80.0	N/A	N/A
PerfoGraph	94.0	90.0	N/A	N/A
MIREncoder	93.6	**93.7**	N/A	N/A
ConTraPh	**95.52**	92.51	**4.44**	**1.76**

4 Comparison with Traditional Tools

We also compared ConTraPh with two traditional program analysis tools: AutoPar (static analyzer) and DiscoPoP (dynamic analyzer). We considered the task of detecting parallel loops. We considered the SP application of NAS Parallel Benchmark and extracted 40 loops. Here we employ our ConTraPh to detect any possible parallel configuration for a given loop. If ConTraPh finds any suitable configuration, we consider that ConTraPh detected it as a parallel loop, otherwise ConTraPh considers it is a non-parallel loop none. We follow the same for AutoPar and DiscoPoP too. From Table 10 it can be observed that ConTraPh detects more parallel loops than both the traditional tools and it has significantly less overhead than DiscoPoP. This is expected as DiscoPoP is a dynamic analyzer and it executes the code to generate results. The impact will be more severe for applications that require larger execution time as DiscoPoP will need to execute programs, whereas ConTraPh will not be affected that much as it does not need this step. Also, static analyzer AutoPar takes slightly less time to generate results but has much less accuracy than ConTraPh.

5 Correctness and Limitations

ConTraPh is designed as an intelligent assistant to aid users in developing parallel programs but not as a replacement. As any other learning-based model ConTraPh can make mistakes. For the task described in Section 3.1 and 3.6, it can suggest wrong parallel configurations and the user can choose to accept or disregard the suggestions from ConTraPh. However, from the results, we believe it is fair to say that as ConTraPh has very high accuracy (around 96%) in most of the cases, the suggestions will be accurate, and it will help users to write proper parallel codes with minimal effort. Regarding tasks in Section 3.2, 3.3, 3.4, and 3.7 even if ConTraPh suggests wrong configurations, it will only lead to worse runtimes but not faulty programs. However, as the performance of ConTraPh is also good in each of these tasks, we believe it is fair to say that ConTraPh suggestions most of the time will lead to optimized programs with better runtimes.

Table 10: Comparing Parallelism Detection Accuracy of AutoPar, DiscoPoP and ConTraPh on SP application of NAS benchmark. In total, there are 40 loops. 25 Parallel (P) and 15 Non-parallel (NP)

Tool	Accuracy(%)	Detected P loops	Detected NP loops	Time Taken
AutoPar	67.5%	12	15	**235 milliseconds**
DiscoPoP	80%	17	15	89600 milliseconds
ConTraPh	**95%**	**23**	15	315 milliseconds

6 Scalability Testing

We also perform scalability testing on the parallel configurations predicted by ConTraPh. Eight applications (total of 90 loops as discussed in 3.1) of NAS Parallel Benchmark are used for this experiment. First, the OpenMP configurations suggested by ConTraPh are collected, and then complete OpenMP clauses are generated by invoking the GPT-4 by incorporating ConTraPh predictions in the prompt as discussed in Section 3.5. We compare with the ground-truth parallel configurations, and if there is a mismatch, we use the sequential version of the loop such that ConTraPh does not gain any unfair advantage and also the correctness of the program is preserved. Also, we verify that the outcome matches with the original output for the applications parallelized using ConTraPh. We use the 'CLASS A' input that comes along with NPB for evaluating the applications. Each application (both the sequential and parallelized versions) is executed five times. Then, the speedup is calculated by dividing the average sequential runtime by the average parallel runtime. Five thread configurations, 2, 4, 8, 16, and 32, are considered. Also, a computing node containing an Intel Xeon Gold 6152 processor with 22 cores, 2.1 GHz of Base Frequency (with max frequency up to 3.7 GHz), and 30.25 MB of L3 cache with DDR4 memory is used. Figure 7 shows that EP scales the best (achieves around 8X speedup) as the application is 'embarrassingly parallel' and trivial to scale. Applications like LU, FT, and BT are computationally intensive and usually outweigh the thread communication

overheads; hence, it can be observed that they scale up to 32 threads and achieve speedups up to 4.4X, 3.1X, and 2.7X, respectively. All other applications also achieve speedup where the minimum speedup gain is observed for SP (1.2X for four threads). However, these applications (SP, IS, MG, CG) do not scale well for higher thread counts as they involve frequent inter-thread communication and synchronization. These operations introduce significant overhead, thereby creating a bottleneck that limits scalability.

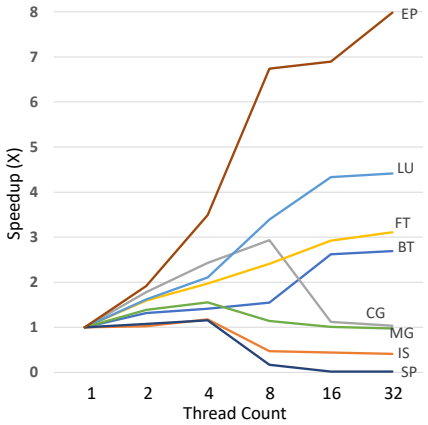

Figure 7: Scalability testing of NAS benchmark applications for the parallel configurations predicted by ConTraPh.

7 Related Works

Pragformer [14] uses transformer-based [37] architecture for identifying parallel patterns. It only considers 2 patterns and uses source text of codes and ASTs for detecting parallelism. Graph2Par [8] proposed augmented AST and GNNs for identifying different parallel patterns. There are also other works like [31, 32] that use GNNs for identifying parallelism in sequential programs. Authors of [31] used GCN along with CFGs for identifying parallel regions in source code, whereas authors of [32] used AST along with control and data flow for the same problem. However, these works only consider identifying parallelizable code regions but do not generate any suggestions regarding optimizations. MIREncoder [11] uses ProGraML to model programs for performance optimization. However, they use the regular GNNs with cross-entropy-based loss as the basis for their learning model. Recently, [35] proposed an IR-based fine grain program representation PerfoGraph, which is tailored for performance optimization tasks like parallelism discovery, parallel pattern detection etc. PerfoGraph achieved SOTA results in several performance optimization tasks. The works [18, 24] tried to tackle to problem of automatically generating parallel codes. However, both of these works are only applicable to very few parallel configurations, which results in bottlenecks in performance improvement. Finetuning of LLMs is also performed in the study of [7]; however, authors indicated that the accuracy remained quite low for parallel configuration prediction.

There are some static (Pluto [5], Rose [27]) and Dynamic analyzers (DiscoPoP [19]), which offer autonomous parallelization, however, these traditional tools may miss a lot of optimization opportunities and may also be computationally expensive as pointed out by authors [8, 31, 32]. Most of the above-mentioned works either focus on only detecting parallelizable sections in code or try to parallelize using very limited parallel configurations. Also, none of the works explore the optimization opportunities that lie within different parallel implementation styles of the same program.

There are some works that use contrastive learning for different code-related tasks like code search and code vulnerability prediction. For example, Cocosoda [33] uses source code and text query for code search, authors of [9] use dropout masks on value-flow graph embedding for generating contrastive samples in programs for software vulnerability prediction. Authors of [39] use contrastive learning on source code and flattened AST pair for vulnerability prediction. HELoC [40] only considers hierarchical AST along with contrastive learning for code classification tasks. However, none of these works combine high-level textual information of source code with the fine-grained low-level Intermediate Representation (LLVM-IR) of code for contrastive learning.

ConTraPh addresses this gap. To the best of our knowledge, we are the first to propose an MSC loss-based contrastive learning approach using multi-view program representation for performance optimization that supports a wide range of parallel configurations and can optimize among different parallel configurations. Our approach combines contrastive learning with source codes and a more granular IR-based program graph with data dependencies, data flow, and explicit control flow constructs. This multi-view setup enhances the model's ability to capture complex program semantics, supporting optimizations like parallelization and instruction scheduling.

8 Conclusion and Future Work

In this paper, we propose ConTraPh, a contrastive learning-based approach that learns by utilizing distance-based MSC loss using multiple views of the same program. We evaluated ConTraPh on six parallelization and performance optimization-related downstream tasks and demonstrated that it surpasses the state-of-the-art models in almost all the tasks. We also showed that ConTraPh is flexible and can be

used to support different parallel programming models. Future work will involve predicting more challenging optimization configurations like optimal thread counts, Vectorization Factor (VF), Interleaving Factor (IF) using ConTraPh.

Acknowledgments

We would like to thank NSF for their generous support in funding this project (#2211982) and (#2422127). In addition, we extend our gratitude to Cisco AI Research and Intel Labs for their generous support in the project. We thank the Research IT team of Iowa State University for providing access to HPC clusters for conducting the experiments of this research project.

References

[1] 2024. GPT-3.5 Models. https://platform.openai.com/docs/models/gpt-3-5-turbo. [Online; accessed 3-april-2024].

[2] 2024. GPT-4 Models. https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo. [Online; accessed 3-april-2024].

[3] 2024. LLVM IR Based Transformation Flags. https://llvm.org/docs/Passes.html#transform-passes. [Online; accessed 3-april-2024].

[4] 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).

[5] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. 2008. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer.

[6] 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.

[7] Le Chen, Arijit Bhattacharjee, Nesreen Ahmed, Niranjan Hasabnis, Gal Oren, Vy Vo, and Ali Jannesari. 2024. Ompgpt: A generative pre-trained transformer model for openmp. In *European Conference on Parallel Processing*. Springer, 121–134.

[8] 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).

[9] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 519–531.

[10] 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.

[11] Akash Dutta and Ali Jannesari. 2024. MIREncoder: Multi-modal IR-based Pretrained Embeddings for Performance Optimizations. *arXiv preprint arXiv:2407.02238* (2024).

[12] 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.

[13] 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.

[14] Re'em Harel, Yuval Pinter, and Gal Oren. [n. d.]. Learning to Parallelize Source Code via OpenMP with Transformers. ([n. d.]).

[15] Niranjan Hasabnis. 2022. GitRank: a framework to rank GitHub repositories. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 729–731.

[16] Hao-Qiang Jin, Michael Frumkin, and Jerry Yan. 1999. The OpenMP implementation of NAS parallel benchmarks and its performance. (1999).

[17] Nicholas Johnson and Adrian Jackson. 2013. The EPCC OpenACC benchmark suite. In *Exascale Applications and Software Conference*.

[18] Tal Kadosh, Niranjan Hasabnis, Vy A Vo, Nadav Schneider, Neva Krien, Mihai Capota, Abdul Wasay, Nesreen Ahmed, Ted Willke, Guy Tamir, et al. 2023. Domain-Specific Code Language Models: Unraveling the Potential for HPC Codes and Tasks. *arXiv preprint arXiv:2312.13322* (2023).

[19] Zhen Li, Rohit Atre, Zia Huda, Ali Jannesari, and Felix Wolf. 2016. Unveiling parallelization opportunities in sequential programs. *Journal of Systems and Software* 117 (2016), 282–295.

[20] Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, and Felix Wolf. 2015. DiscoPoP: A profiling tool to identify parallelization opportunities. In *Tools for High Performance Computing 2014: Proceedings of the 8th International Workshop on Parallel Tools for High Performance Computing, October 2014, HLRS, Stuttgart, Germany*. Springer, 37–54.

[21] 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.

[22] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[23] Yiqian Liu, Noushin Azami, Avery VanAusdal, and Martin Burtscher. 2023. Choosing the Best Parallelization and Implementation Styles for Graph Analytics Codes: Lessons Learned from 1106 Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[24] Quazi Ishtiaque Mahmud, Ali TehraniJamsaz, Hung D Phan, Nesreen K Ahmed, and Ali Jannesari. 2023. Autoparllm: Gnn-guided automatic code parallelization using large language models. *arXiv preprint arXiv:2310.04047* (2023).

[25] Gleison Souza Diniz Mendonça, Chunhua Liao, and Fernando Magno Quintão Pereira. 2020. AutoParBench: a unified test framework for OpenMP-based parallelizers. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–10.

[26] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, USA, 311–318. doi:10.3115/1073083.1073135

[27] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.

[28] Tal Reiss and Yedid Hoshen. 2023. Mean-shifted contrastive loss for anomaly detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 2155–2162.

[29] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE]

[30] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv*

preprint arXiv:2308.12950 (2023).

[31] Yuanyuan Shen, Manman Peng, Shiling Wang, and Qiang Wu. 2021. Towards parallelism detection of sequential programs with graph neural network. *Future Generation Computer Systems* 125 (2021), 515–525.

[32] Yuanyuan Shen, Manman Peng, Qiang Wu, and Guoqi Xie. 2023. Multi-graph learning for parallelism discovery in sequential programs. *Concurrency and Computation: Practice and Experience* 35, 9 (2023), e7648.

[33] Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Cocosoda: Effective contrastive learning for code search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2198–2210.

[34] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[35] 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 (NeurIPS)* 36 (2024).

[36] 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.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

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

[39] Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. 2024. Combining Structured Static Code Information and Dynamic Symbolic Traces for Software Vulnerability Prediction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[40] Xiao Wang, Qiong Wu, Hongyu Zhang, Chen Lyu, Xue Jiang, Zhuoran Zheng, Lei Lyu, and Songlin Hu. 2022. Heloc: Hierarchical contrastive learning of source code representation. In *Proceedings of the 30th IEEE/ACM international conference on program comprehension*. 354–365.

[41] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. (2023). https://arxiv.org/abs/2302.05527