ParaGraph: Weighted Graph Representation for Performance Optimization of HPC Kernels

Ali TehraniJamsaz[1], Alok Mishra[2], Akash Dutta[1], Abid M. Malik[3], Barbara Chapman[2,3], Ali Jannesari[1]
[1]*Iowa State University, Ames, Iowa, USA*
{tehrani, adutta, jannesar}@iastate.edu
[2]*Hewlett Packard Enterprise, Milpitas, California, USA*
{alok.mishra, barbara.chapman}@hpe.com
[3]*Stony Brook University, Stony Brook, New York, USA*
{amalik, bchapman}@cs.stonybrook.edu

***Abstract*—GPU-based HPC clusters are attracting more scientific application developers due to their extensive parallelism and energy efficiency. In order to achieve portability among a variety of multi/many core architectures, a popular choice for an application developer is to utilize directive-based parallel programming models, such as OpenMP. However, even with OpenMP, the developer must choose from among many strategies for exploiting a GPU or a CPU. This paper introduces a new graph-based program representation for optimization of OpenMP applications. The originality of this work lies in the augmentations of Abstract Syntax Trees (ASTs) and the introduction of edge weights to account for loop and condition information. We evaluate our proposed representation by training a Graph Neural Network (GNN) to predict the runtime of OpenMP code regions across CPUs and GPUs. Various transformations utilizing collapse and data transfer between the CPU and GPU are used to construct the dataset. The trained model is used to determine which transformation provides the best performance. Results indicate that our approach is effective and has normalized RMSE as low as 4×10^{-3} to at most 1×10^{-2} in its runtime predictions.**

***Index Terms*—OpenMP, HPC, offloading, program representation**

I. Introduction

Over the years, the increase in the number of on-chip cores has led to significant improvement in the performance of parallel code. To exploit this increased computation capacity, applications have been readily modified using Pthreads or OpenMP constructs. In the last decade, General Purpose Graphic Processing Units (GPGPUs) have gained popularity. HPC platforms will continue to support more accelerators, but given the difficulties in utilizing and configuring even one accelerator, using and configuring multiple heterogeneous accelerators will become increasingly difficult in the future.

On the other hand, utilizing GPUs effectively imposes challenges that require re-engineering the code and applications. It is exceptionally challenging and burdensome for developers to create applications for extremely heterogeneous platforms with multiple devices. The recent emergence of tools and programming models aims to automate this process of application adaptation to heterogeneous platforms. One of the most popular parallel programming models, OpenMP [1], aims to make the process of developing parallel programs that can run on different architectures simpler. Despite this, optimizing a code to use the OpenMP directives correctly is still tedious for large and complex applications.

Recent advances in Deep Learning (DL) have enabled researchers to apply DL to a wide range of software engineering problems and challenges. Programs need to be represented in a suitable representation that not only serves as input to DL models but also exposes necessary features. In this paper, we propose $ParaGraph$, a graph-based program representation that aims to expose critical characteristics (e.g., the flow of a loop) of HPC applications. To the best of our knowledge, existing program representations are not specifically designed to expose and represent the characteristics of parallel HPC applications. As a result, DL models built on top of these representations cannot model features inherent to parallel codes effectively. We apply our program representation to the task of runtime prediction across CPUs and GPUs. Previous works, such as [2], have relied on feature engineering to predict the runtime of kernels on GPUs. However, feature engineering requires expert knowledge. For a fast-evolving field such as HPC, always relying on expert intervention for such feature engineering is not realistic. There is a need for an adaptable approach that can automatically extract such features. Our proposed approach addresses this gap and leverages Graph Neural Networks (GNNs) to model the code graphs generated by $ParaGraph$. Experimental results show that our model can predict the runtime of HPC kernels with a very low error rate (at most 1×10^{-2} in terms of normalized RMSE), confirming the efficacy of our strategic approach.

In summary, the main contributions of the paper are as follows:

- An innovative runtime prediction model that facilitates portability across heterogeneous HPC platforms.
- A novel representation of HPC kernels for deep learning models that highlights their HPC characteristics.
- Constructing a new data set consisting of variations of HPC kernels supporting both CPUs and GPUs.
- Evaluation of the proposed HPC kernel representation by predicting runtime for different devices, including both CPUs and GPUs, and outperforming the state-of-the-art approach.

II. Background and Related Work

In this section, related works and some background for program representation, OpenMP parallelism, and its tools are discussed.

A. Program Representation

Recently, with breakthroughs in Machine Learning (ML) and specifically Deep Learning (DL), researchers have been applying data-driven approaches to various software engineering tasks and challenges, ranging from code comment generation [3] to compiler optimizations [4].

Recently, graph-based program representations have been proposed that can model different program flows such as control flow, data flow, etc. Allamanis et al. [5] proposed an AST-based program graph representation to model programs to two tasks of variable misuse detection and variable name suggestion.

Cummins et al. in [4] provided a lower-level graph representation based on LLVM intermediate representation for solving compiler optimization tasks.

While these representations are effective for downstream tasks, such as simple algorithm classification, to the best of our knowledge, there does not exist a representation tailored toward representing the characteristics of HPC kernels and important control statements such as *parallel* loops and if-statements. Other representations, such as the one in [2], are engineered for specific accelerators and can not be applied to various accelerators and platforms. Considering the heterogeneity of HPC platforms and the increasing demand for portable applications that can be run on different accelerators, the need for a program representation that is independent of the underlying accelerator has never been more crucial.

B. OpenMP GPU Offloading

Developers need to ensure program portability across diverse GPU architectures and compilers, particularly in heterogeneous HPC systems. One approach to ensure portability across heterogeneous architectures is using a directive-based programming paradigm, such as OpenMP, the de-facto standard for parallel programming in C/C++ and Fortran. OpenMP intends to move to extremely heterogeneous architectures [6] and has supported GPU offloading from specification 4.0. Unfortunately, even with OpenMP, optimizing large-scale applications remains a challenging task.

The "`omp parallel for`" pragma alone will only parallelize a code for CPUs; it won't offload the computation to a GPU. We expect a high level of coarse grain parallelism on a platform like GPU. The amount of parallelism that a GPU can use is constrained by this design. Figure 1 illustrates how OpenMP `teams` and `distribute` directives create additional levels of parallelism. At the start of a target region, only one team and one member thread are active. The `teams distribute` directive distributes the full loop iteration space among all available teams. We utilize the combined directive "`teams distribute parallel for`" to distribute the iteration space of one loop among teams and threads inside a team when there is just one level of parallel loops or when the outer loop has sufficient parallelism. `teams` are used to group threads, and `distribute` enables a team group to be scheduled to run in a loop.

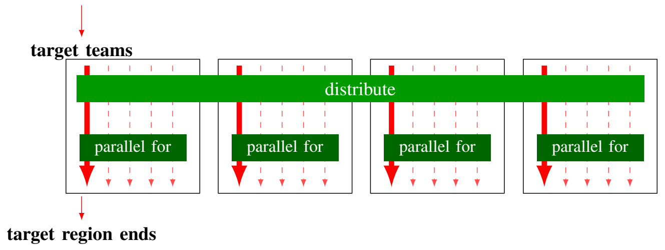


Fig. 1: OpenMP target teams distribute parallel for.

There are several frameworks being worked on right now that will automatically assist application developers in handling severe heterogeneity. Hand-tuned cost functions are extensively employed currently; however, calculating optimization costs requires a deeper understanding of the underlying hardware. Since cost functions are crucial and manual tuning is time-consuming, compiler engineers are investigating Machine, and Deep Learning approaches as a means of automating this process.

C. ML in Compiler

Recently, a lot of research has been done on how to use learning-based techniques in compilation as well. Early work exploiting ML in compilers primarily explored its use to help optimize sequential programs. However, its application to the task of optimizing parallel programs has recently attracted attention due to the prevalence of multi-core platforms and, more recently, heterogeneous systems [7], [8]. Mirka *et.al.* [9] devised a decision-tree-based approach to predict the scheduling policy for an OpenMP parallel region. Also, Denoyelle *et.al.* [10] uses machine learning techniques to optimize OpenMP programs for scheduling policies and the number of threads. Tree and graph-based features have been used by Malik *et.al.* [11], who present a graph-based approach for feature representation. Learning-based techniques were used to build classifiers to determine whether to offload OpenCL code [12] and to select a clock frequency at which the processor should operate [13]. A high level of accuracy was reported; however, the benefits could not be quantified as the work did not attempt to generate modified code. They also explored regression techniques to build curve fitting models to search for the Pareto frontier for work partitioning among processors [14] or a trade-off of energy and performance [15].

In COMPOFF [2], the authors provide a proof of concept for using fully connected feed-forward network model to make better decisions in offloading a kernel to a GPU using OpenMP.

D. OpenMP Advisor

OpenMP Advisor [16] is a compiler tool that enables OpenMP code offloading to a GPU via Machine Learning. The Advisor is divided into three major modules: Kernel Analysis, Cost Model, and Code Transformation. The Kernel Analysis module recommends various variants for a given application, and the Code Transformation module generates codes for those variants. In our work, we use the code transformation module of OpenMP Advisor to generate various kernels for training our model. OpenMP Advisor uses COMPOFF [16] to identify the kernels that are most suitable for offloading by predicting their runtime. But `COMPOFF` has some limitations. It requires figuring out how many operations are contained within a kernel, which is a challenging task in and of itself. The Advisor's current functionality is restricted to GPUs. In this work, we will compare $ParaGraph$ to `COMPOFF`.

III. ParaGraph

In this section, we present $ParaGraph$, a graph representation of programs that captures characteristics related to HPC kernels. $ParaGraph$ aims to represent programs using information available at compile time to enable deep learning models to reason over HPC kernels and supports heterogeneous platforms. For instance, it can help users decide which accelerator would be a better fit for a particular kernel by predicting the runtime of that kernel statically. $ParaGraph$ leverages the compiler Abstract Syntax Tree (AST) and incorporates additional information, such as edge weights, to count for loops and if-statements. For example, $ParaGraph$ presents *loops* by special edges conveying the order in which the loop condition and its body execute iteratively. Additionally, $ParaGraph$ adds weights to the edges to expose how many times each region or scope will be encountered during the execution of a program if this information is available at compile time. $ParaGraph$ code representation can be easily modeled by Graph Neural Networks. Our experimental results show that, in performance optimization, this representation is quite effective and outperforms the state-of-the-art approach.

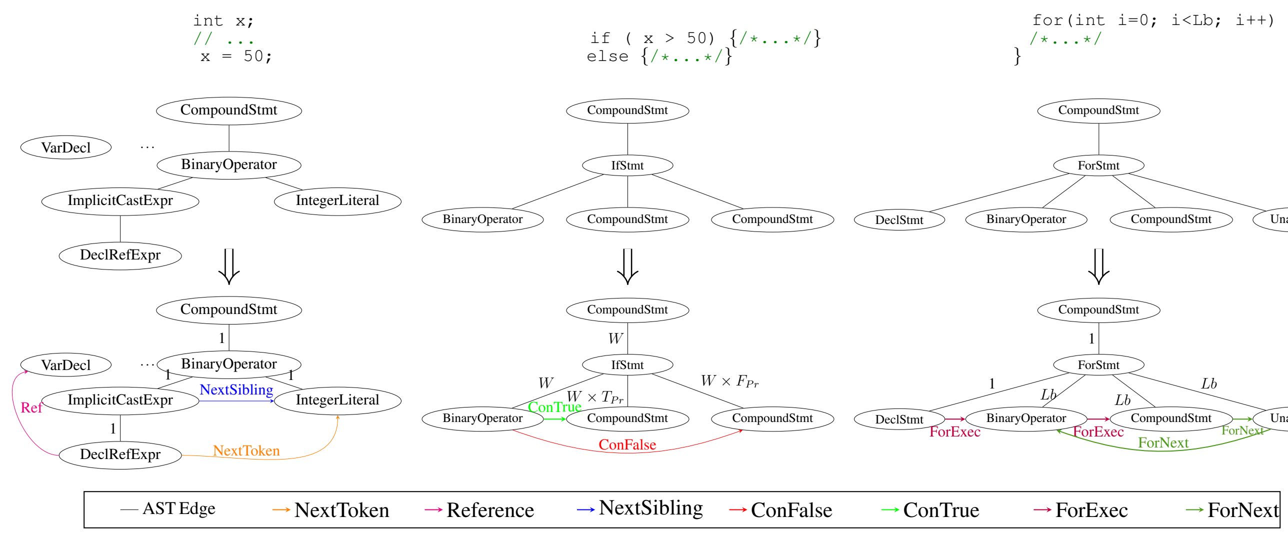
A. ParaGraph Construction

1) Abstract Syntax Tree: $ParaGraph$ graph structure is built on top of AST. In this study, we use `Clang`[1] to parse and compile `C/C++` programs. ASTs contain two types of nodes: *non-terminal* and *terminal*. Non-terminal nodes are often called *syntax nodes*, whereas terminal nodes are called *syntax tokens*. Nodes in ASTs have a parent-child relation. $ParaGraph$ augments the AST by introducing additional edges and attributes to convey control and data flow information. The following subsections will provide more details on these augmentations.

2) Augmenting Abstract Syntax Tree: ASTs typically provide structural and syntactic information about a program. Although this information can be useful for neural networks to learn the characteristics of programs to some extent, it has been shown before [17], [5] that adding additional attributes and information, especially from a compiler point of view, boosts the learning capabilities of deep learning models. $ParaGraph$ introduces the following additional edges to AST (shown in Figure 2).

[1]https://clang.llvm.org/

- **NextToken:** By default there is no order imposed among the *syntax tokens*. However, from a compiler's perspective, syntax tokens have an order. To present this information in a graph, $ParaGraph$ introduces a new edge type called `NextToken`. `NextToken` connects each syntax token to the syntax token on its right side. This edge type is shown in orange in Figure 2 (the AST on the left).
- **NextSib:** AST's edges show a parent-child relationship between nodes. On the other hand, compilers intrinsically consider an order among children nodes. For example, a binary operator such as division has two children. In an AST, the left child is always the numerator, and the right child is the denominator. Therefore, it is necessary to show the order of children. `NextSib` connects each syntax node to its sibling on the right (the blue edge in Figure 2).
- **Ref:** In Clang's AST, referenced variables are presented by `DecRefExpr` nodes. These nodes are terminal and do not have children. $ParaGraph$ adds `Ref` edges (shown in pink in Figure 2) connecting a `DecRefExpr` node to where the corresponding variable is defined. This edge will convey information about where a declared variable is used throughout the graph.
- **ForNext, ForExec:** Loops are typically shown as `ForStmt` nodes in Clang's AST. `ForStmt` nodes usually have four children. The first child initializes the loop counter, the second is the loop's condition. The third child is a `CompoundStmt`, which presents the body of a loop. Lastly, the fourth child modifies the loop counter such as incrementing it by one. The relationship between these four children exposes critical characteristics that are known by compilers. We add `ForExec` edges, which connect the first child (counter initialization) to the second child (loop condition) and also connect the second child to the third child (loop body), as shown in the right side of Figure 2. In fact, `ForExec` edges show the flow of executing the next iteration of the loop. We connect the third child (loop's body) of the `ForStmt` node to the fourth child (loop counter modifier) via a new type of edge called `ForNext`. `ForNext` represents the information related to whether the next iteration of the loop needs to be executed or not, whereas `ForExec` represents the next execution of the loop. The fourth child (loop counter modifier) is followed by the second child (loop condition), which checks if the next iteration is going to execute or if the loop has ended. Therefore, the fourth child is connected to the second child with a *ForNext* edge as shown in Figure 2.
- **ConTrue, ConFalse:** If statements in the ASTs are shown by `IfStmt` nodes. `IfStmt` nodes typically have three children. First child presents the condition; the second child is the body of the `if` condition and the third child is the body of the `else` part. To present this information, we connect the first child to the second child through a new type of edge called `ConTrue` to show that the flow moves to the second child if the `if statement` is true. On the other hand, the first child is connected to the third child via a `ConFalse` edge. This edge shows that if the condition of



```
int x;
// ...
 x = 50;
```

```
if ( x > 50) {/*...*/}
else {/*...*/}
```

```
for(int i=0; i<Lb; i++) {
 /*...*/
}
```

Fig. 2: Modification to AST to create Augmented AST for ParaGraph.

`if` statement is not met, the flow moves to the third child.

– **Child:** Child edges are normal AST edges that present a parent-child relationship among the nodes in AST.

3) Weighted Edges: In the previous section, we explained how $ParaGraph$ augments the AST by introducing new types of edges to better present programs and exposes information about how `loops` or `if statements` are executed. However, there is still some information missing in our representation. For instance, whenever we encounter a loop in a program, the loop's body could be executed multiple times, or the number of times each branch of an `if statement` is executed is not the same. As a result, the constructed graph needs to be further augmented to include this missing information. To solve this issue, $ParaGraph$ considers adding weights to the edges. Weights of edges are calculated based on the region and edge type. Weights are added only to `Child` edges since these are AST edges, and a compiler uses the AST nodes and child edges to transform an AST to lower-level for machine execution. For illustration purposes, some edge weights are shown in Figure 2. We add the weights as follows:

– **Default edge weight:** By default, we initially add a weight of 1 to each `Child` edge assuming, for now, that each statement in the code executes only once, and once a statement executes, the control moves to the next statement.

– **Loops:** To expose the number of iterations in our graph representation, we first observe the number of iterations in a loop and then multiply the edge weights by that number. $ParaGraph$ uses the information available at compile time to determine the loop bound (Lb). Moreover, the workload of each thread is also implicitly taken into account if the loop scheduling is static. This is done by dividing the number of iterations by the number of threads.

– **If statements:** The nodes and edges under one branch of an if statement versus the other one will not have the same number of executions. Therefore, there is a need for justification of edges of `if` statements. To illustrate the execution of branches, $ParaGraph$ can apply probabilities to branches in the graph. Depending on the user's choice, different branch prediction tools can be used to estimate the probability of each branch. Once the probabilities are retrieved, $ParaGraph$ applies them to the edge weight of the `if-statements` (T_{Pr}: the probability that the condition is true, F_{Pr}: the probability that the condition is false as shown in Figure 2). Such that, if the edge weight of the $CompoundStmt$ before the $IfStmt$ is W, then the edge weight of the $True$ branch will be $W \times T_{Pr}$ and that of the $False$ branch will be $W \times F_{Pr}$. As a result, the branch that has a higher probability will have a larger edge weight, whereas the branch will a lower probability will have a smaller edge weight.

$ParaGraph$, in the future, can be extended to other conditional statements. One example of such extension is when dealing with a `Switch` statement that has '`n`' number of cases or a chain of '`n`' `if-else-if statements`. In such scenarios, our representation can divide the edge weights by '`n`' for each case. The ablation study results show that edge weights help the models reason better over the kernels. It is worth noting that because all these augmentations are applied statically, therefore their overhead is minimal.

B. Adapting GNNs

$ParaGraph$ can be used by existing GNN models, which are a type of neural network that can operate over graphs for downstream tasks. Typically, a graph is shown as Equation 1, where V is a set of nodes and E contains an adjacency matrix. Elements of E are either 1 or 0 to show whether an edge exists between two nodes.

$$G = (V, E) \qquad (1)$$

We extend the AST for our new representation by including new edges like `NextToken`, `NextSib`, and other types that

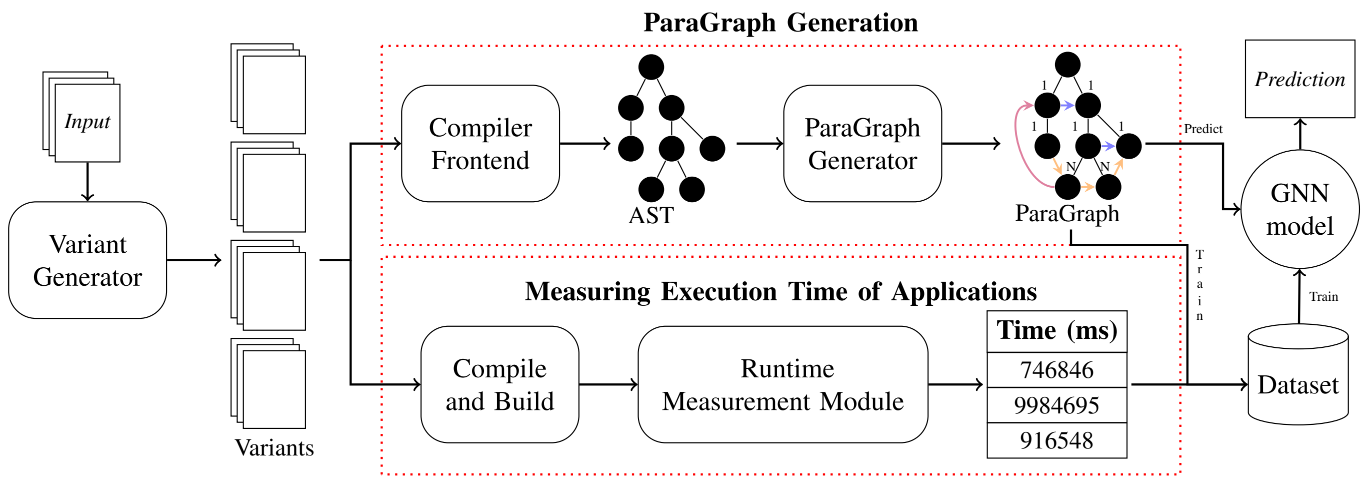


Fig. 3: Workflow of ParaGraph.

were explained earlier. We formally define $ParaGraph$ as follows:

$$ParaGraph = (V, E, T, W) \tag{2}$$

Where V and E are previously defined in Equation 1 and $T \in Z^+$ represents the type of the edges (such as `NextToken`, `NextSib` etc). $W \in Z^+$ presents the weight which is zero for any edge type other than `Child`.

These additional edges and attributes are all added to the graphs statically. Additionally, the overhead of generating these AST-based graphs is minimal. We will later show in our ablation study that our simple modeling technique benefits significantly from the attributes and information presented in $ParaGraph$.

We adapt Relational Graph Attention Networks (RGAT) [18] and use $ParaGraph$ representation as input to train a model for predicting runtime of applications on different accelerators. In RGAT, attention logits are computed per each edge type. In the result section, we will see that with $ParaGraph$, the model has a very small prediction error and outperforms the current state-of-the-art approach.

Figure 3 shows the overall workflow of our GNN-based pipeline for runtime prediction. The first step is preparing different variants of an application for each accelerator used in this study. This step is further explained in detail in the next section. Then, using `Clang`, ASTs are produced, and a series of augmentations, as discussed, are applied to them to construct $ParaGraph$. In order to train a model to predict the runtime of an application, we need to create a dataset with a list of applications and their variant accompanied by their runtime. Therefore, we execute each one of the variants on the specified accelerator and measure the runtime. Lastly, the $ParaGraph$ representation of variants and their runtime are used to train the GNN model. Along with $ParaGraph$s, our feature set also includes the number of teams and threads used for executing an application. These features are also fed along $ParaGraph$ into the GNN-based model for predicting the runtime.

IV. Experiments and Setups

We used two clusters and compilers to test and evaluate our tool. Our experiments were carried out on the ORNL's Summit supercomputing cluster [19] with LLVM/Clang (ver 13.0) and a `nvptx` backend for GPUs, as well as the LLNL's Corona cluster [20] with LLVM/Clang (ver 15.0) and a `rocm` backend for GPUs. For the purposes of this study, we only use one GPU per cluster node.

One of the primary obstacles we faced, as with any other data-driven approach, is the lack of a publicly accessible dataset. Creating a dataset that can be used to train our model was the first step in building a data-driven cost model. Although benchmark suites like Rodinia [21], PolyBench [22], SPEC OMP [23] and the Barcelona OpenMP Task Set (BOTS) [24] are developed to evaluate the performance of parallel applications on various hardware platforms and to compare the performance of various parallel programming models, little effort has been put into OpenMP GPU offloading benchmarking. While useful for assessing and comparing the performance of parallel applications, these benchmark suites are not suitable for investigating the efficacy of a novel program representation or optimization strategy on heterogeneous architectures. There is a lack of OpenMP GPU offloading benchmarks that are publicly accessible. Therefore, as a first step, we create a collection of benchmarks, borrowing from existing benchmarks like Rodinia and adding a few commonly used benchmarks in scientific applications that leverage OpenMP GPU offloading. The goal is to include a broad class of application domains that would cover a spectrum of statistical simulation, linear algebra, data mining, etc. This gave us the opportunity to train and test our proposed program representation in a more realistic and representative setting.

A. Data Collection

There are three parts to our data collection: Code Variant Generation, Graph Generation, and Runtime Collection.

1) Code Variant Generation: The first step was to collect and analyze kernels that are widely used in real-world state-of-the-art HPC setups from multiple application domains. Table I shows nine such benchmark applications that have been used in this paper. From these nine applications, all seventeen unique OpenMP kernels were extracted and used for data collection. For this work, we have only focused on the following six transformations:

- **cpu**: A cpu parallel kernel using `omp parallel for`.
- **cpu_collapse**: In the case of a nested collapsible loop, we collapse it with `omp parallel for collapse(2)` directive.
- **gpu**: A gpu kernel using a combined `omp target teams distribute parallel for` directive. All data is already considered to be present on the GPU.
- **gpu_collapse**: A gpu kernel with nested collapsible loop using `omp target teams distribute parallel for collapse(2)` directive. All data is already considered to be present on the GPU.
- **gpu_mem**: Same as combined gpu offloading, but with data transfer.
- **gpu_collapse_mem**: Same as combined gpu_collapse, but with data transfer.

We used the OpenMP Advisor tool [16] to generate these kernel variants, leading to the creation of 66 distinct kernels. To further augment our dataset, we also varied the levels of parallelism and input data for each of these kernels. Taking into account these modified parameters and the kernel variants, we were able to generate $\approx$ 2000-3000 unique kernels from each application and around 26,000 unique data points across all applications. These steps, so far, are independent of the target architecture. The code kernels, variations in code transformations, variations in degree of parallelism, and data, in reality, create new unique kernels. When compiled to their corresponding ASTs, these are unique in their structure, as each code transformation modifies the AST output by the compiler. Our representation accounts for such differences in the compiled ASTs and adds new additional features relevant to parallel code, as discussed. These design choices are essential to convey to a model/tool enough details about the kernel being modeled.

2) ParaGraph Generation: In this step of our pipeline, we create the $ParaGraphs$ for the unique kernels created in the previous section. As shown in Figure 3, prior to generating the

TABLE I: Benchmark Applications.

Application	**Num Kernels**	**Domain**
Correlation Coefficient [25]	1	Statistics
Covariance	2	Probability Theory
Gauss Seidel Method	1	Linear Algebra
K-nearest neighbors [21]	1	Data Mining
Laplace's Equation [26]	2	Numerical Analysis
Matrix-Matrix Multiplication	1	Linear Algebra
Matrix-Vector Multiplication	1	Linear Algebra
Matrix Transpose	1	Linear Algebra
Particle Filter [21]	7	Medical Imaging

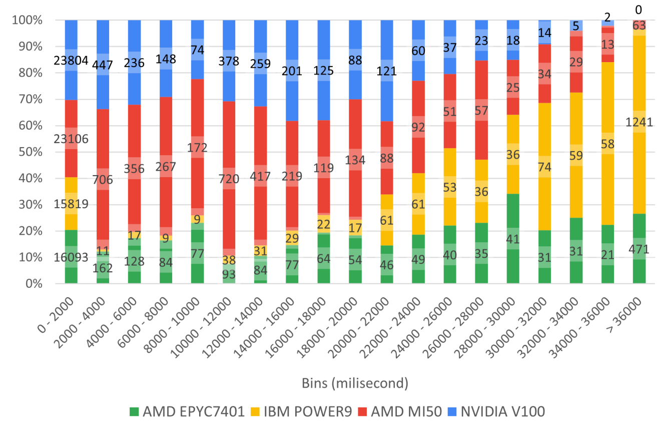


Fig. 4: Distribution of all collected data points.

$ParaGraphs$ for each kernel, they are first compiled to their corresponding ASTs. Each AST is then traversed and updated with additional edges and weights as outlined in Section III-A. As mentioned before, these additional edges and weights are critical for exposing probable execution bottlenecks, such as loops, as well as providing information to the GNN model about data and control flow in the kernel. Typically, edges within a `ForLoop` have larger weights to represent the number of iterations of a loop. Loop bounds are extracted from the AST itself, and branch weights are considered 0.5 for the sake of simplicity.

3) Runtime Collection: Due to the lack of existing datasets targeting OpenMP GPU offloading, our first step was to develop/build such a dataset. The aim of this step was to collect the runtimes of each combination of code kernel, code variant, and other parameters, such as the degree of parallelism and input data used. This dataset has been used for evaluating the strength of our code representation. But it can also be used for research tasks that aim to optimize decision-making or cost functions in the compiler.

We used the OpenMP Advisor tool [16] to collect kernel runtime information. This tool inserts function calls before and after a kernel call to evaluate the wall time of a kernel's execution. Each application with its kernel variants was compiled individually on each cluster with the appropriate compile-time parameters to utilize the respective backends. The compiled binary was then executed on the cluster to capture their corresponding runtime. This led to the collection of around 83k samples across the two clusters with runtimes ranging between 0.024 ms and 737 seconds. Statistics about individual clusters and devices are shown in Table II. Additionally, Figure 4 shows the distribution of collected datapoint on 2 CPUs and 2 GPUs.

These runtimes were used as targets to train our GNN-based regressor model.

B. ParaGraph Model

Post data collection on the Summit and Corona clusters, we begin training our model. We implemented a GNN-based neural network using RGAT [18] as the convolution layers. The inputs to the RGAT layers are the $ParaGraphs$ generated

TABLE II: Statistics of collected data points.

Platform	#Data Points	Runtime Range (ms)	Std. Dev.
Summit			
IBM POWER9 (CPU)	13,023	[0.23 - 736,798]	48,502
NVIDIA V100 (GPU)	26,040	[0.035 - 30,174]	3,708
Corona			
AMD EPYC7401 (CPU)	17,681	[0.024 - 291,627]	16,942
AMD MI50 (GPU)	26,668	[0.448 - 46,913]	4,828

from the code kernels and transformations. For each node, we extract four features from AST: `kind` (e.g, IntegerLiteral), `type` (e.g., int), `opcode` (e.g., ==) and `value` (the value of the node if it is a terminal node). If a feature does not exist for a node, we set that feature to `None`. Edge weights are regarded as the feature of the edges. Our model is implemented using Pytorch-Geometric library with Mean Squared Error as the loss function and Adam [27] as the optimizer. To embed the input graph, the model uses three graph convolution layers based on RGAT, followed by two fully connected layers with `ReLu` activation function. As mentioned, the number of teams and threads are considered as two additional features. Another fully connected layer is used to embed these two features. Finally, the embedding of the graph and the two features are concatenated together and passed through the last fully connected layer for runtime prediction.

The edge weights and the two additional features are normalized using MinMaxScaler. The dataset is split randomly into train-validation sets using a 9:1 ratio. There is no overlap between the train set and the validation set as each *ParaGraph* graph will be placed only in one of them.

V. Results

In the following subsection, we explain the metrics we have used to evaluate the $ParaGraph$ model.

A. Evaluation Metric

To evaluate the performance of $ParaGraph$, we use $RMSE$, which is Root Mean Square Error (Equation 3).

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \hat{x}_i)^2}{N}} \tag{3}$$

Where x_i stands for the runtime of a data point in microseconds, $\hat{x}_i$ is the predicted runtime by $ParaGraph$, and N is the total number of samples.

Since the range of runtime differs across platforms, the normalized version of $RMSE$ is also considered. Normalized $RMSE$ is calculated by dividing the $RMSE$ by the distance between the minimum and maximum runtime. We also use relative error (i.e., absolute error divided by the range of runtime) to report the error rate.

B. Experimental Results

Table III shows the experimental results for each accelerator and CPU. We have NVIDIA V100, AMD MI50, and IBM POWER9 with 22 cores and AMD EPYC 7401 with 24 cores. As shown, the $RMSE$ values range from 280 (ms) to 4325 (ms). The reason why we have different $RMSE$ values, such as 4325 (ms) for POWER9 accelerator, lies in the fact that the runtime dispersion differs across the accelerators. Standard Deviation in Table II gives us some insights on how dispersed the collected data are. Moving on to Normalized-RMSE, which is independent of the range of runtime, we see $ParaGraph$ has relatively the same error across accelerators; thus, it can be applied to different accelerators.

TABLE III: Experimental Results.

Platform	RMSE (ms)	Norm-RMSE
Summit		
IBM POWER9 (CPU)	4325	6×10^{-3}
NVIDIA V100 (GPU)	280	9×10^{-3}
Corona		
AMD EPYC7401 (CPU)	968	4×10^{-3}
AMD MI50 (GPU)	510	1×10^{-2}

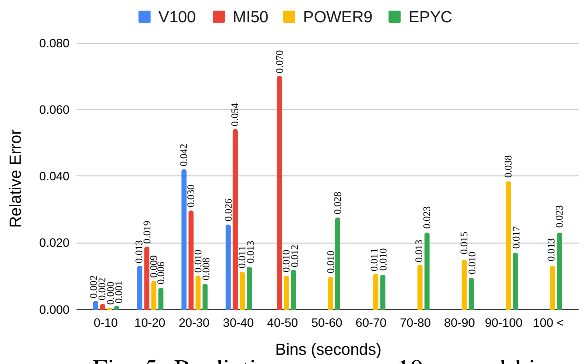


Fig. 5: Prediction error per 10-second bins.

To further analyze our results, we have calculated a relative error per bins of 10 seconds. Figure 5 shows 11 bins for all four accelerators. Each bin has a 10-second range except the last bin. The figure shows that the relative error is small across different bins and accelerators (less than %10), meaning that our $ParaGraph$ model has stable behavior across varying problem sizes, accelerators, and kernels.

Moreover, we analyze how stable the model is during the training process in Figure 6. The figure shows validation $RMSE$ for the two GPUs and two CPUs. In the first few epochs, as expected, the $ParaGraph$ model is not very stable, resulting in fluctuations in the $RMSE$; however, as the model is trained further, it is able to better extract and reason about the features from the code representation and reduce $RMSE$ value per each epoch and ultimately converges.

Lastly, we calculate the average relative error per application to evaluate the prediction error rate of the $ParaGraph$ model for all types of applications. Figure 7 shows the error rate of each application. As can be seen, the model can indeed make accurate predictions for a wide variety of applications resulting in a low error rate. This proves that the model trained on the code $ParaGraphs$ is generalizable and not biased towards specific kernels/applications. On the AMD MI50 GPU, the Laplace data was corrupted during collection. Consequently, neither this study nor the training process includes that data.

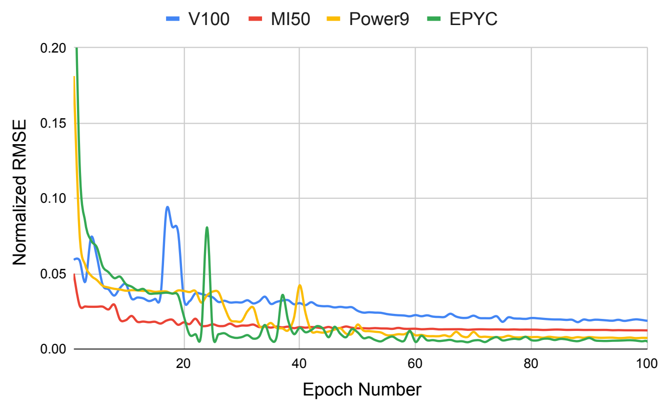


Fig. 6: Normalized RMSE per each epoch.

C. Ablation Study

ParaGraph representation, as explained in Section III, is built by applying a series of major augmentations on top of AST. In this section, we quantify the impact of these augmentations. First, we consider the AST itself without additional edges and weights; we call it Raw AST. Then, we add additional edges and edge types to the AST and name it Augmented AST. Lastly, we have *ParaGraph* that contains both additional edges and also edge weights. Table IV shows the results of the ablation study. We see that Raw AST results in the highest error for all CPUs and GPUs included in this study. Adding new edges and introducing new types of edges (Augmented AST) improve the prediction to some extent. For example, the $RMSE$ of V100 drops from 2114 (ms) to 786 (ms) with the addition of these new edges.

One of the key characteristics of our proposed program representation is the edge weights. Edge weights convey essential information about how often different regions of the AST execute. Therefore, we see quite a good improvement in $RMSE$ when edge weights are added. For instance, the $RMSE$ for V100 is further improved to 280 (ms).

We analyze the addition of edges and their weights further to see how the training process of the *ParaGraph* model is affected by these augmentations.

Figure 8 shows how the model behaves during training. This figure depicts the $RMSE$ value per each epoch for Raw AST,

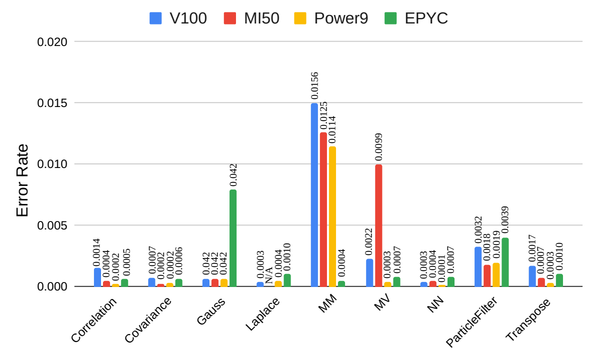


Fig. 7: Error rate per each application.

TABLE IV: RMSE of training with and without edges' weight.

Platform	Raw AST	Aug AST	ParaGraph
Summit			
IBM POWER9 (CPU)	27593	26860	**4325**
NVIDIA V100 (GPU)	2114	786	**280**
Corona			
AMD EPYC7401 (CPU)	11911	9633	**968**
AMD MI50 (GPU)	2888	1177	**510**

Augmented AST, and *ParaGraph* on the MI50 accelerator.

Using only the Raw AST without any augmentations, which means having only one edge type, the model is able to learn some characteristics of the applications and reduce the $RMSE$ per epoch; however, this reduction in $RMSE$ is not significant. Augmented AST contains eight different types of edges. We see the addition of these edges destabilized the training process of the model. In the first few epochs, the model is challenged to learn different relations between the nodes. However, eventually, after several epochs, the prediction of the model is stabilized, and it achieves $RMSE$ of 1177 (ms). Once edges of the Augmented AST are augmented with weights, thus *ParaGraph* is constructed, and we see further improvements in the model's predictions. Although the validation $RMSE$ fluctuates in the initial epochs, it ultimately converges with a considerably smaller error.

D. Comparison with State-of-the-art Tool

To the best of our knowledge, $COMPOFF$ [2] is the

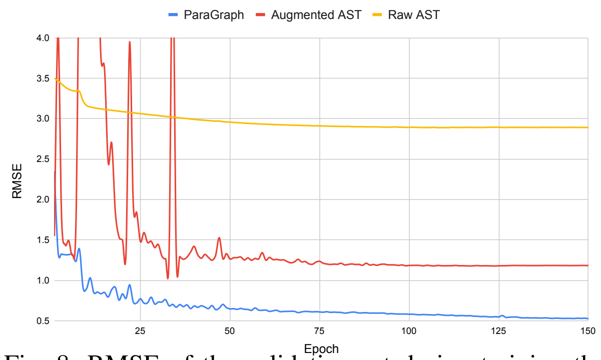


Fig. 8: RMSE of the validation set during training the GNN models on MI50 data points.

```
void mm_kernel_gpu_mem(double (*A)[N2], double
↪   (*B)[N3], double (*C)[N3], FILE *fp)
{
#pragma omp target teams distribute parallel
↪   for map(to: A[0:N1][0:N2], B[0:N2][0:N3])
↪   map(from: C[0:N1][0:N3])
  for(int i=0; i<N1; i++) {
    for(int j=0; j<N3; j++) {
      double sum = 0.0;
      for (int k = 0; k < N2; k++)
        sum = sum + A[i][k] * B[k][j];
      C[i][j] = sum;
    }
  }
}
```

Listing 1: Matrix Multiplication OpenMP Offloading.

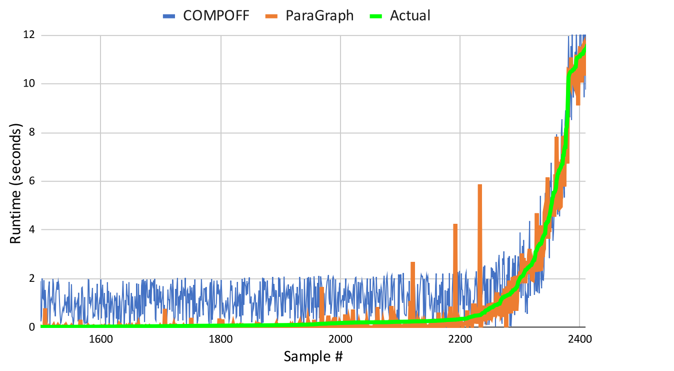


Fig. 9: ParaGraph and COMPOFF prediction on NVIDIA V100 as compared to the actual runtime.

only state-of-the-art OpenMP GPU offloading cost model. We further compare the results from $ParaGraph$ with those of $COMPOFF$. As mentioned in Section II-D OpenMP Advisor uses $COMPOFF$ for predicting runtime for OpenMP kernels. While OpenMP Advisor eventually needs a cost model that can forecast for all possible underlying architectures, Due to the feature selection part of $COMPOFF$, it is only suitable for GPU executions as features are engineered toward GPU executions. In contrast, $ParaGraph$ enables GNN models to learn the features automatically. Thus it can be applied to any other architecture given enough data points, as we saw $ParaGraph$ was applied to predict the runtime of both CPUs and GPUs in our experiments. Here, We compare $ParaGraph$ against $COMPOFF$ on NVIDIA V100 GPU.

Figure 9 demonstrates a correlation between the actual and predicted runtime from $COMPOFF$ and $ParaGraph$. The results for $COMPOFF$ are represented by blue dots, while those for $ParaGraph$ are represented by orange dots. As we can see, $ParaGraph$ demonstrates a stronger correlation between the predicted and actual runtime. Listing 1 shows an implementation of Matrix Multiplication kernel being offloaded to GPU. $N2$ and $N3$ are specified during compilation time. In our setup, it takes 10.5 seconds to execute this kernel on a V100 GPU with $N2 = 7600$ and $N3 = 9600$. $ParaGraph$ predicts the execution time for this kernel to be 10.3 seconds, having a small error, whereas $COMPOFF$'s prediction is 8.5 seconds with a higher error.

As demonstrated in Figure 11, $COMPOFF$ (blue) demonstrates a higher error rate for smaller runtime kernels, but as the runtime increases, this error rate decreases. However, for all kernels, $ParaGraph$ has a lower error rate on average and it does not suffer a high error rate for smaller runtime kernels.

We also compare $ParaGraph$ and `COMPOFF` in terms of percent error. Due to the very high percent error of `COMPOFF` (especially for smaller runtimes), we present the result in log scale in Figure 10. As it can be seen, on average $ParaGraph$ has a lower error rate. On larger runtime, $ParaGraph$ is on par with `COMPOFF`, however, sometimes it has a slightly higher error rate. We posit the reason is that more data points with larger runtimes are needed.

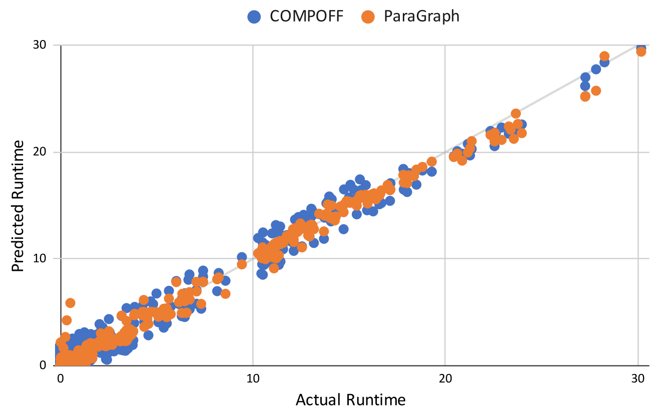


Fig. 10: Logscale of percent error of ParaGraph and COMPOFF.

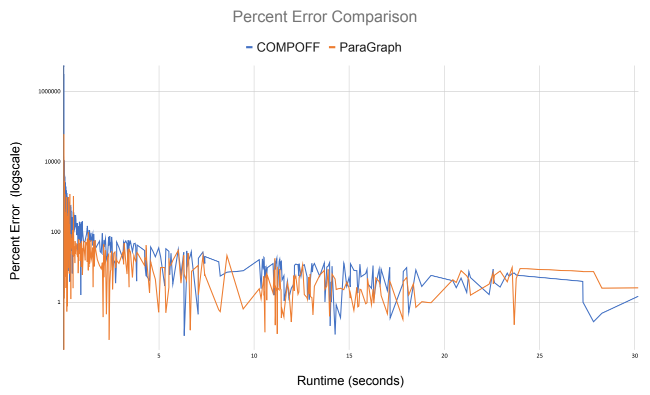


Fig. 11: Comparison of ParaGraph and COMPOFF on predicting runtimes on NVIDIA V100 for each data point.

E. Discussion

As demonstrated by our experimental results, $ParaGraph$ has exceptional static modeling capability, allowing it to make accurate predictions of application runtimes across heterogeneous devices. $ParaGraph$ is a compile-time tool limited to information available during compilation only. Other complementary prediction tools can be used in situations when some information is unavailable to estimate the missing data. Handling missing runtime inputs is outside the scope of $ParaGraph$, as it primarily focuses on compile-time code variant selection. In such circumstances, $ParaGraph$ can leverage specialized tools such as loop-bound or branch prediction algorithms to gain significant insights beyond static analysis.

Combining $ParaGraph$'s compile-time code variant selection with complementary prediction tools (e.g., loop-bound, branch prediction, and profilers) results in a comprehensive runtime prediction strategy that is adaptive to diverse contexts. This dynamic hybrid methodology enhances predictions when runtime inputs are unavailable during compilation. Through their collaboration, $ParaGraph$ and other prediction tools can improve the overall efficacy and usability across heterogeneous architectures.

VI. Conclusion and Future Work

In this paper, we proposed $ParaGraph$, a novel way of representing HPC kernels. $ParaGraph$ includes some of the insights of the compiler by adding new edges to the AST. We evaluated $ParaGraph$ on a set of applications for four different architectures. It achieved a low error rate, highlighting the effectiveness of $ParaGraph$. The hardware-architecture independence of $ParaGraph$ is a significant advantage, allowing it to be applied to a broader spectrum of architectures than previous approaches. This feature is especially crucial given the growing demand for portable kernels and the prevalence of heterogeneous HPC platforms.

In this work, we used $ParaGraph$ to predict the execution time of a given kernel for the OpenMP GPU offloading problem. We plan to explore and analyze how $ParaGraph$ can help other OpenMP optimization strategies, such as predicting SIMD stride, scheduling strategies, loop chunk size, etc. Another interesting research area to explore is to use $ParaGraph$ and capture parallelism for other parallel programming models, such as OpenACC, Kokkos, HIP, etc.

In the future, we also plan on looking into the potential benefits of using a hybrid approach that combines both static and dynamic features to more accurately model application runtime. We can potentially improve the accuracy of our predictions for applications with unstable runtime behaviors by incorporating dynamic features such as performance counters, which can provide real-time information about an application's execution. We believe that this hybrid approach will result in better prediction results and will allow us to model a wider range of applications more effectively.

Acknowledgments

This research was funded by the National Science Foundation (NSF) under grant number 2211982. Also, we would like to thank the Research IT team[2] of Iowa State University for their continued support in providing HPC clusters.

References

[1] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[2] A. Mishra, S. Chheda, C. Soto, A. M. Malik, M. Lin, and B. Chapman, "COMPOFF: A compiler cost model using machine learning to predict the cost of openmp offloading," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 30-June 3, 2022*. IEEE, 2022.

[3] A. Ciurumelea, S. Proksch, and H. C. Gall, "Suggesting comment completions for python using neural language models," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 456–467.

[4] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. O'Boyle, and H. Leather, "Programl: A graph-based program representation for data flow analysis and compiler optimizations," in *International Conference on Machine Learning*. PMLR, 2021, pp. 2244–2253.

[5] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[6] M. A. Heroux, R. Thakur, L. McInnes, J. S. Vetter, X. S. Li, J. Aherns, T. Munson, and K. Mohror, "Ecp software technology capability assessment report," Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), Tech. Rep., 2020.

[7] Y. Kim, J. Lee, J.-S. Kim, H. Jei, and H. Roh, "Comprehensive techniques of multi-gpu memory optimization for deep learning acceleration," *Cluster Computing*, vol. 23, no. 3, pp. 2193–2204, 2020.

[8] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[9] M. Mirka, G. Sassatelli, and A. Gamatié, "Online learning for dynamic control of openmp workloads," in *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. IEEE, 2020, pp. 1–6.

[10] N. Denoyelle, B. Goglin, E. Jeannot, and T. Ropars, "Data and thread placement in numa architectures: A statistical learning approach," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[11] A. M. Malik, "Automatic static feature generation for compiler optimization problems," in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2011, pp. 769–778.

[12] S. Dublish, V. Nagarajan, and N. Topham, "Poise: Balancing thread-level parallelism and memory system performance in gpus using machine learning," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 492–505.

[13] A. Iranfar, W. S. De Souza, M. Zapater, K. Olcoz, S. X. de Souza, and D. Atienza, "A machine learning-based framework for throughput estimation of time-varying applications in multi-core servers," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2019, pp. 211–216.

[14] D. Grewe and M. F. O'Boyle, "A static task partitioning approach for heterogeneous systems using opencl," in *International conference on compiler construction*. Springer, 2011, pp. 286–305.

[15] H. Sayadi, "Energy-efficiency prediction of multithreaded workloads on heterogeneous composite cores architectures using machine learning techniques," *arXiv preprint arXiv:1808.01728*, 2018.

[16] A. Mishra, A. M. Malik, M. Lin, and B. Chapman, "Openmp advisor: A compiler tool for heterogeneous architectures," in *19th International Workshop on OpenMP, IWOMP 2023, Bristol, UK, September 12–15, 2023*. Springer, 2023.

[17] M. Allamanis, "Graph neural networks in program analysis," in *Graph Neural Networks: Foundations, Frontiers, and Applications*. Springer, 2022, pp. 483–497.

[18] D. Busbridge, D. Sherburn, P. Cavallo, and N. Y. Hammerla, "Relational graph attention networks," *arXiv preprint arXiv:1904.05811*, 2019.

[19] ORNL, "Oak Ridge Leadership Computing Facility - Summit supercomputing cluster," 2017. [Online]. Available: https://www.olcf.ornl.gov/summit/

[20] L. L. N. Laboratory, "LLNL - Corona," 2019. [Online]. Available: https://hpc.llnl.gov/hardware/compute-platforms/corona

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[22] L.-N. Pouchet and T. Yuki, "Polybench/c," *URL: http://web. cse. ohio-state. edu/~ pouchet*, vol. 2, 2016.

[23] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "Specomp: A new benchmark suite for measuring parallel computer performance," in *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools, WOMPAT 2001 West Lafayette, IN, USA, July 30–31, 2001 Proceedings*. Springer, 2001, pp. 1–10.

[24] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *2009 international conference on parallel processing*. IEEE, 2009, pp. 124–131.

[25] P. Schober, C. Boer, and L. A. Schwarte, "Correlation coefficients: appropriate use and interpretation," *Anesthesia & Analgesia*, vol. 126, no. 5, pp. 1763–1768, 2018.

[26] A. K. Mitra, "Finite difference method for the solution of laplace equation," *Department of aerospace engineering Iowa state University*, 2010.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

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