2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-3766-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPS54959.2023.00060

# Power Constrained Autotuning using Graph Neural Networks

Akash Dutta
*Iowa State University*
Iowa, USA
adutta@iastate.edu

Jee Choi
*University of Oregon*
Oregon, USA
jeec@uoregon.edu

Ali Jannesari
*Iowa State University*
Iowa, USA
jannesari@iastate.edu

***Abstract*—Recent advances in multi and many-core processors have led to significant improvements in the performance of scientific computing applications. However, the addition of a large number of complex cores have also increased the overall power consumption, and power has become a first-order design constraint in modern processors. While we can limit power consumption by simply applying software-based power constraints, applying them blindly will lead to non-trivial performance degradation. To address the challenge of improving the performance, power, and energy efficiency of scientific applications on modern multi-core processors, we propose a novel Graph Neural Network based auto-tuning approach that (i) optimizes runtime performance at pre-defined power constraints, and (ii) simultaneously optimizes for runtime performance and energy efficiency by minimizing the energy-delay product. The key idea behind this approach lies in modeling parallel code regions as flow-aware code graphs to capture both semantic and structural code features. We demonstrate the efficacy of our approach by conducting an extensive evaluation on 30 benchmarks and proxy-/mini-applications with 68 `OpenMP` code regions. Our approach identifies `OpenMP` configurations at different power constraints that yield a geometric mean performance improvement of more than 25% and 13% over the default `OpenMP` configuration on a 32-core Skylake and a 16-core Haswell processor respectively. In addition, when we optimize for the energy-delay product, the `OpenMP` configurations selected by our auto-tuner demonstrate both performance improvement of 21% and 11% and energy reduction of 29% and 18% over the default `OpenMP` configuration at Thermal Design Power for the same Skylake and Haswell processors, respectively.**

***Index Terms*—Auto-tuning, OpenMP, GNN, Power constraint**

## I. Introduction

High-performance computing (HPC) systems have exploded in both capacity and complexity over the past decade, and this has led to substantial improvement in performance of various scientific applications. However, more complex larger systems consume more power, and in the absence of expensive cooling solutions, increased power consumption leads to higher operational temperature and inefficient resource utilization (via higher static power, shorter device lifespan, and more). As a result, power consumption has become a first-order hardware design constraint for modern multi- and many-core systems. Unfortunately, focusing on hardware advancements for reducing power consumption is insufficient, as inefficient usage of the underlying hardware due to poor parallel coding practices may negate any hardware improvements.

Many software solutions currently exist for controlling power. At the processor level, vendor-provided tools can be used to artificially lower power consumption. For example, power consumption can be controlled in recent Intel processors using the Running Average Power Limit (RAPL) interface [1], which ensures that an application does not exceed a predefined power budget. However, a common drawback of a fixed power budget is that it *slows down* execution by lowering the processor clock, and this can have adverse effects on real-time or time-bound applications. At the data-center level, a common approach to reducing power consumption is through over-provisioning (i.e., have more hardware available than can be powered simultaneously at any time) and constraining the power limit for each node [2]. In such a setting, a static algorithm for distributing power across nodes may lead to *degraded throughput*, and a more sophisticated approach that adjusts the execution dynamically is required to harness the full potential of the underlying system.

One strategy to address both scenarios is to adjust the execution of the application directly, such that they meet some user-specified (e.g., individuals or data-centers) performance and/or power constraints. This will allow users to tailor their application to domain-specific environments (e.g., edge or mobile computing) or design scheduling policies for data-center power management. `OpenMP`, as the de-facto parallel programming model for intra-node parallelism, provides a number of tunable parameters that highly influence code execution, which makes it highly suitable for this purpose. While there is already a large body of work targeting performance tuning, there are only a few studies that target power. In addition, due to the large configuration search space for `OpenMP` on modern multi- and many-core processors, most of these studies require multiple executions to determine the optimal configuration [3]–[7], which is both time consuming and resource intensive.

As a motivating example, we consider the *ApplyAccelerationBoundaryConditionsForNodes* kernel from the `LULESH` [8] proxy application. On a 16-core dual-socket Haswell processor with a Thermal Design Power (TDP) of 85W, an exhaustive search of the `OpenMP` configuration space yields the highest speedups of 7.54×, 2.11×, 1.80× and 1.67× over the typical (or default) `OpenMP` configuration at power constraints of 40W, 60W, 70W and 85W, respectively. However, *none* *of these* OpenMP *configurations lead to the highest energy efficiency*. The most energy-efficient execution occurs at a power constraint of 60W using a OpenMP configuration that leads to a greenup (i.e., greenup = $\frac{Energy_{old}}{Energy_{new}}$ [9]) of $3.89\times$, but a speedup of $0.95\times$ (i.e., a *slowdown*) over the typical OpenMP configuration at TDP (85W). This contradicts the commonly held belief of *race-to-halt* [10] (i.e., the idea that the lower energy consumption occurs at the highest speedup), and shows that optimizing for time and optimizing for energy may not yield the same OpenMP configuration. In addition, for applications where a slowdown is unacceptable, we can simultaneously optimize for time and energy by targeting the energy-delay product (EDP) metric [11]. Through an exhaustive search through the OpenMP configurations space, we observe that minimizing EDP yields a speedup of $1.64\times$ and a greenup of $2.7\times$, at a yet another OpenMP configuration and power constraint.

In summary, optimizing for performance, power, and energy consumption all require different strategies for identifying the optimal OpenMP configuration, and optimizing for one metric (e.g., performance) does not necessarily optimize for another (e.g., energy). To this end, we propose a graph neural network (GNN)-based technique that can be used to (i) identify OpenMP configurations at prescribed power constraints that maximizes performance and (ii) optimize for the *energy-delay product* to identify configurations for both energy-efficient and performant execution.

In this study, OpenMP code regions are first transformed to a flow-aware graphical representation. These code graphs are then modeled by a GNN, and used for predicting the best configurations for the appropriate target. In contrast to prior studies, we use only these code graphs (i.e., static features) as inputs to our model, which does not require *expensive code execution*. The benefit of using a deep learning (DL)-based approach is that it automatically helps reduce the search space exploration by aggressively pruning non-beneficial points in the search space.

The works in [6], [7] studied the impacts of power constraints and OpenMP configurations on time and energy and are, to the best of our knowledge, most similar to the problem considered in this paper. To demonstrate the effectiveness of our static approach, we compare our results against a Bayesian Optimization based tuner BLISS [5], and a search-based tuner OpenTuner [4]. Through this study, we propose two separate approaches for tuning performance and energy/power. The first approach aims to identify the tuning configuration that can produce the fastest execution at a predefined power constraint. The second approach looks at both time and energy as target metrics and aims to optimize for both at the same time by identifying configurations that lead to the lowest *energy-delay product*. The key contributions of our work are as follows:

- We build an RGCN network to model flow-aware OpenMP code region graphs that captures both semantic and structural features of code regions, and is portable across different architectures.
- We build an auto-tuning framework that identifies OpenMP configurations yielding near optimal execution times at different power constraints. We achieve a geometric mean speedup of $1.33\times$ and $1.15\times$ over default OpenMP configurations at four power constraints across 30 applications on Skylake and Haswell systems.
- Our DL-based framework also optimizes for both time and energy simultaneously by minimizing the EDP. We achieve geometric mean speedup of $1.27\times$ and $1.12$, and greenup of $1.40\times$ and $1.22\times$ respectively on Skylake and Haswell, over default OpenMP configurations running at TDP (i.e., no power constraint).
- We compare our framework against the state-of-the-art BLISS [5] tuner and OpenTuner [4] and demonstrate better performance without the need for executing code.

## II. Background and other Related Works

This section outlines ideas and works relevant to this paper.

### A. Autotuning for Performance Optimization

Autotuning is a widely-used technique employed in compiler and runtime optimization tasks for performance enhancements. Automated techniques of autotuning have been the focus of research over the past several decades. Autotuners improve upon brute-force approaches by using/proposing several search space optimization techniques which largely reduce the tuning overhead. Algorithm-based autotuners employ multiple techniques for such tasks. Simplex based optimization algorithms were used in ActiveHarmony [3]. More recent algorithm-based tuners such as OpenTuner [4] have used various techniques, including Nelder-Mead, Torczon hill-climbers, AUC Bandit for optimizing search spaces.

A more recent trend has been the use Bayesian optimization for search space optimization and pruning. Works such as ytopt [12], HiPerBOt [13], BLISS [5] have successfully adapted Bayesian optimization ideas to autotuning tasks. These works usually define a (or sets of) probabilistic surrogate model(s) which is usually a surrogate of the true objective function. These surrogate functions are faster to compute and are usually less resource intensive than previous approaches.

Machine learning (ML) has also been used frequently used for such tuning tasks. Classical ML techniques such as Decision Trees, Random Forests, SVMs have been used for compiler-based tuning tasks [14]. Deep learning or Artificial Neural Networks (ANN) have also found favor amongst researchers. Works such as [15], [16] have effectively used deep learning for various tuning tasks. Recently, a few works [17], [18] have also used reinforcement learning for targeted optimization tasks.

In this paper, we use a deep learning based approach for our tuning tasks, and compare our results with results obtained from OpenTuner and BLISS.

### B. Power Constraining and Energy Usage Reduction

As mentioned in Section I, power and energy are nowadays first-order design considerations. Power constraining is a software based easy-to-use technique that can be used to limit the power supply to various system components. Starting with the SandyBridge $\mu$architecture, Intel introduced the RAPL software tool that enables power/energy monitoring and power capping through a simple interface. The power to several subsytems of the processor, such as memory, DRAM, CPU, etc. can be controlled via RAPL.

Most autotuning works in existing literature, however, do not consider such power constraints in their work. A few papers have focused on dynamic voltage frequency scaling (DVFS) and dynamic concurrency throttling (DCT) techniques for improving energy efficiency [19]–[21]. Wang et al. in [22] proposed using CPU clock modulation and concurrency throttling for improving the energy efficiency of `OpenMP` loops. In [23], Nandamuri et al. analyzed the performance and energy conumption of `OpenMP` programs under various conditions using `OpenMP` Runtime API. The impact of CPU parameters on performance and energy for `OpenMP` dense linear algebra kernels was presented in [24]. Rountree et al. in [25] provided a first insight into the impacts of power capping or constraints on power and performance. Patki et al. in [26] outlined how overprovisioning hardware and hardware enforced power bounds leads to improved performance. Bari et al. in [6] propose ARCS with the goal of automatically selecting best runtime configurations for `OpenMP` parallel regions at specified power constraints and in [7] analyzed the impact of power constraints on performance and energy consumption on five NAS benchmarks. To the best of our knowledge, the works in [6], [7] are the closest to this paper. In contrast to [6], [7], our approach uses an AI-assisted technique based on GNNs to identify `OpenMP` runtime configurations and power constraints.

### C. Static Code Representations for Deep learning

Deep learning is being increasingly used in modeling code for various tasks [27]. However, the use of deep learning necessitates the use of a code representation capable of capturing its inherent features. A lot of prior studies have represented programs as a sequence of lexical tokens [28]. But, these fail to capture the structured nature of programs. To overcome this, representations capturing syntactic as well as semantic features have been proposed [27], [29] .

These methods, however, often do not take into account control, data, or call flows in the program. PROGRAML [28] is a tool that represents the semantic and structural features of code in a flow-aware multi-graph. These multi-graphs have a vertex for each instruction and control-flow edges between them. Data flow is represented by separate vertices for variables and constants and associated data-flow edges to instructions. Call flow is represented by edges between callee functions and caller instruction vertices. We use this tool to transform code region IRs to their corresponding graphs.

### D. Graph Neural Network based Code Modeling

Recent advances in deep learning have now enabled the application of DL on data generated from non-Euclidean space [30]. The relations and dependencies between objects in such data can more readily be represented as a graph. Graph Neural Networks (GNNs) were proposed as a means of modeling such data. Most such networks use *message passing* to update the embeddings in neighboring nodes in a graph. Graph Convolutional Networks (GCNs) are a form of GNNs aimed at generalizing the common *sliding window* convolution operation on grid data in regular Convolutional Neural Networks to graphs [30]. A GCN network updates its node representation by aggregating the features from the node's neighbors along with the node. Similar to CNNs, GCNs stack multiple convolutional layers to extract high-level node representation. We use Relational Graph Convolutional Network (RGCN), a variation of GCN, to model our program graphs. RGCNs were proposed to enable networks to better model large-scale relational data [31]. RGCNs differ from GCNs in that they work with relation specific transformations annotated by the type and direction of edges. RGCNs accumulate transformed feature vectors through a normalized sum.

Recently, researchers have started applying GNN-based techniques to the task of code modeling [16], [32]. This, in most cases, involves compiling source code into their graph forms and then using these code graphs as inputs to the GNN models. The greatest progress in this field has been seen in software engineering tasks such as code clone detection, code summarization, etc. Previous works such as [33], [34] have achieved state-of-the-art results in various tasks using GNN-based code modeling. However, GNNs have rarely been used for the task of parallel code modeling with energy and performance optimizations in mind. The works in [16], [32] are examples from a very small set that have used GNNs for the purpose of modeling parallel code with specific performance optimizations in mind.

In this work, we have used RGCNs to model the code graphs of `OpenMP` code regions. The results in the following sections clearly shows that such an approach produces good results in comparison to general purpose autotuners.

## III. The PnP Auto-Tuner: A GNN based Power and Performance Tuner

In this section, we outline our two-pronged approach to tuning performance and power. We consider two scenarios with real-world implications: i) Because of cost and energy considerations, clusters and data-centers must usually work under strict power budgets. However, constraining power directly impacts performance by limiting the power delivered to hardware components. Therefore, assuming no code changes or compiler optimizations, tuning available runtime parameters becomes essential for improving application performance. ii) It is of utmost importance in most HPC systems to reduce energy consumption. This has a direct monetary and environmental impact. However, as shown in Section I, simply optimizing for energy, can potentially lead to slower executions. Therefore, we must optimize for a metric that considers both energy and performance. To this end, we target the multi-obejctive metric *energy-delay product (EDP)*. We use GNNs to build a model

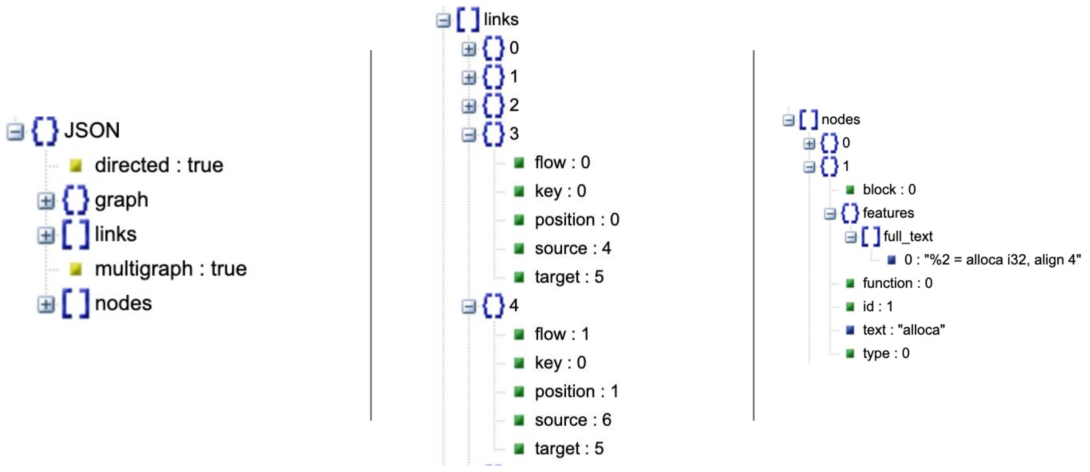


Fig. 1: High-level view of code graphs used in this paper in JSON format. The left image shows the overall structure of a graph. The figure in the middle shows the edge features in the graph. The *flow* attribute denotes the type of program flow. The right-most image shows the node level feautures (each IR instruction forms a node).

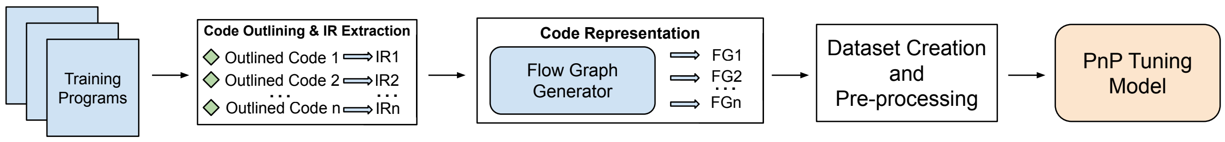


Fig. 2: PnP Tuner Pipeline: An overview of tasks in our GNN based power and performance tuner

that will be used for the aforementioned tasks. The inputs to the GNNs are code flow graphs of `OpenMP` regions. Using such graphs allows us to model the semantics and structure of source code. These convey relevant information to the model about the code region being tuned. We refer to these input code graphs as static features, as these are obtained statically without any code executions. An overview of this pipeline is shown in Figure 2, and outlined in the following paragraphs.

## A. Representing the Code

In this study, we aim to optimize `OpenMP` code regions. These code regions are usually the primary computational bottlenecks in such applications. Instead of focusing on individual loops inside these parallel regions, we aim to optimize the parallel region as a whole for larger performance improvements. Tuning sub-regions within an `OpenMP` code region adds additional overhead. Switching between configurations can improve the performance of each sub-region (loops for example), but can degrade the performance of each `OpenMP` region and the application as a whole. The benchmark applications are initially compiled to their intermediate representations (IR). Compiling `OpenMP` code to its corresponding IR automatically encloses the parallel region in an outlined function. We use the `llvm-extract` tool to extract the outlined parallel region. As shown in Figure 2, to represent the code regions in a form usable by DL models, we use PROGRAML [28] to obtain the corresponding graph embeddings. Each code graph has a structure as shown in Figure 1. The features for each node in the graph and the features for each edge are shown in the Figure. The type of *flow* in these graphs are used to denote the different relations for our RGCN model.

## B. Configuring the Search Space

One of the primary motivations behind using a DL technique for this work was to develop a method that can work with large search spaces easily. Unlike most existing auto-tuners, which have to extensively execute programs to identify the best configurations, our DL-based framework will not need to execute programs. For the proposed DL approach to scale well to unseen code and inputs, it is necessary to feed the model with code graphs with enough variability. Along with variability in considered parallel code regions, it is essential to model the effect of various tuning parameters on these code regions. Different configurations impact code execution by affecting the load balancing and cache behavior, which in turn impacts performance.

As our goal is to target performance optimization and energy efficiency, we must simultaneously consider the impact of power constraints and `OpenMP` parameters on code executions. To this end, we have defined a search space (shown in Table I) with 504 valid configurations. In addition, the default `OpenMP` configurations for each of the four power limits have also been considered as valid configurations leading to a total of 508 configurations. The search space used in this study has been selected based on ideas presented by Bari et al. in [7].

TABLE I: Search space for performance and power tuning on Skylake and Haswell nodes.

| Search Space | Parameter Values |
|---|---|
| Power Limits | 75W, 100W, 120W, 150W (Skylake) |
| | 40W, 60W, 70W, 85W (Haswell) |
| Number of threads | 1, 4, 8, 16, 32, 64 (Skylake) |
| | 1, 2, 4, 8, 16, 32 (Haswell) |
| Scheduling Policy | STATIC, DYNAMIC, GUIDED |
| Chunk Sizes | 1, 8, 32, 64, 128, 256, 512 |

### C. Power Constraining and Dataset Creation

In this work, we used the *Variorum* [35] tool for constraining power levels on each of the experimental systems. We used Variorum APIs to interface with RAPL and device MSRs to constrain power to the values described in Table I.

To validate our hypothesis, we chose to work with multiple OpenMP applications with varied complexity. These OpenMP regions consists of parallel regions with simple do-all loops to regions with multiple loops with varying levels of nesting and diverse programmatic constructs. We have worked with 25 applications from the PolyBench suite [36], and mini and proxy applications XSBench [37], RSBench [38], miniFE [39], miniAMR [40], Quicksilver [41], and LULESH [8] with combined total of 68 OpenMP regions.

At each power level, parallel OpenMP regions in all considered applications were executed for each runtime configuration in Table I and default OpenMP configurations (all threads, static scheduling, and compiler defined chunk sizes) on each system. The execution times obtained as such are then analyzed to identify the best configuration for each code region. The best configurations are used as labels during training.

### D. Performance and Power Modeling

This section outlines our GNN-based approach towards performance and power optimizations. We propose two tuning scenarios with different objectives:

- In the first scenario, we aim to identify the OpenMP configuration that lead to the fastest executions at a given power constraint.
- In the second scenario, we aim to identify both the OpenMP configuration and the power level that minimizes the EDP. By minimizing the EDP, we hope to improve the execution time *and* energy efficiency in comparison to default OpenMP configuration at TDP.

*1) Code Graph Modeling using GNNs:* For both scenarios, the code modeling technique is similar. Modeling code graphs allows us to model code semantics and structure. Analyzing code structure allows us to better capture the interdependence between code blocks. Simply looking at code as a sequence of text does not afford this information. The code graphs generated in Section III-A are initially passed through a GNN network for modeling the code graphs. Specifically, Relational Graph Convolutional Networks (RGCNs) are used as these allow modeling relation specific features. Each code graph consists of three types of edges denoting the type of flow (Section III-A). The type of edges are used as edge features during modeling. For each node in a graph, the node features are the type of node, and the associated IR code block. Before modeling, the code region IRs are used to generate an embedding. This embedding maps IR text to tensors. These tensors are then passed to the model as node features along with the type of the node. Based on these features, the GNN layers model these by passing "messages" between neighboring nodes, aggregation, and subsequent weight updations [42]. The output tensors from the GNN layers then fed into fully connected neural network layers with the aim of identifying the best configurations.

*2) Power Constraint Specific Auto-tuning:* As noted in Section I, one way of meeting power consumption goals is to enforce a specific power constraint. Such power constraints can help limit the power drawn by a node or its subsystems. However, simply using default OpenMP runtime configurations at different power constraints for code execution may lead to performance degradation, as well as increased energy usage from static power. Therefore, we aim to identify those configurations that lead to speedups at predefined power constraints. We propose a DL based technique for power-constrained auto-tuning. As outlined in Section III-D1, we use the flow-aware code graphs obtained from the parallel code region IRs as inputs to the RGCN layers of our network. As shown in Figure 2, the RGCN layers model each such graph and feeds the output into a fully connected (dense) network. The dense layers acts as a classifier and are trained as such with the target of predicting the best configuration for a given OpenMP code region.

*3) Optimizing Energy and Time:* For nodes and systems without any predefined power constraint, time and energy optimization are still of primary importance. However, simply optimizing for performance *or* energy neglects the other criteria. Thus, in this section, we propose using power constraints as a tuning parameter along with the available OpenMP runtime configurations for joint optimization of performance and power. Simply using execution time or energy savings for identifying such configurations is not enough. Thus, we use the *energy-delay product (EDP)* metric [11] as a more accurate measure of the impact of different configurations on code performance. In this work, we assign equal importance to time and energy and use the metric $E * T$, where $E$ represents the energy consumption, and $T$ represents the execution time for a parallel code region.

We again use the modeled code graphs from Section III-D1 as the static feature inputs to our model for this experiment and train our model with a target of optimizing the EDP. As in the previous subsection, a fully connected neural network serves as a classifier to identify the best configurations for tuning EDP. Using a DL-based approach for identifying the best one out of 508 possible configurations is especially beneficial, as such models are efficient at automatically pruning the under performing configurations. This is in stark contrast to brute-force approaches, where the tuning cost would explode with increasing search space complexity.

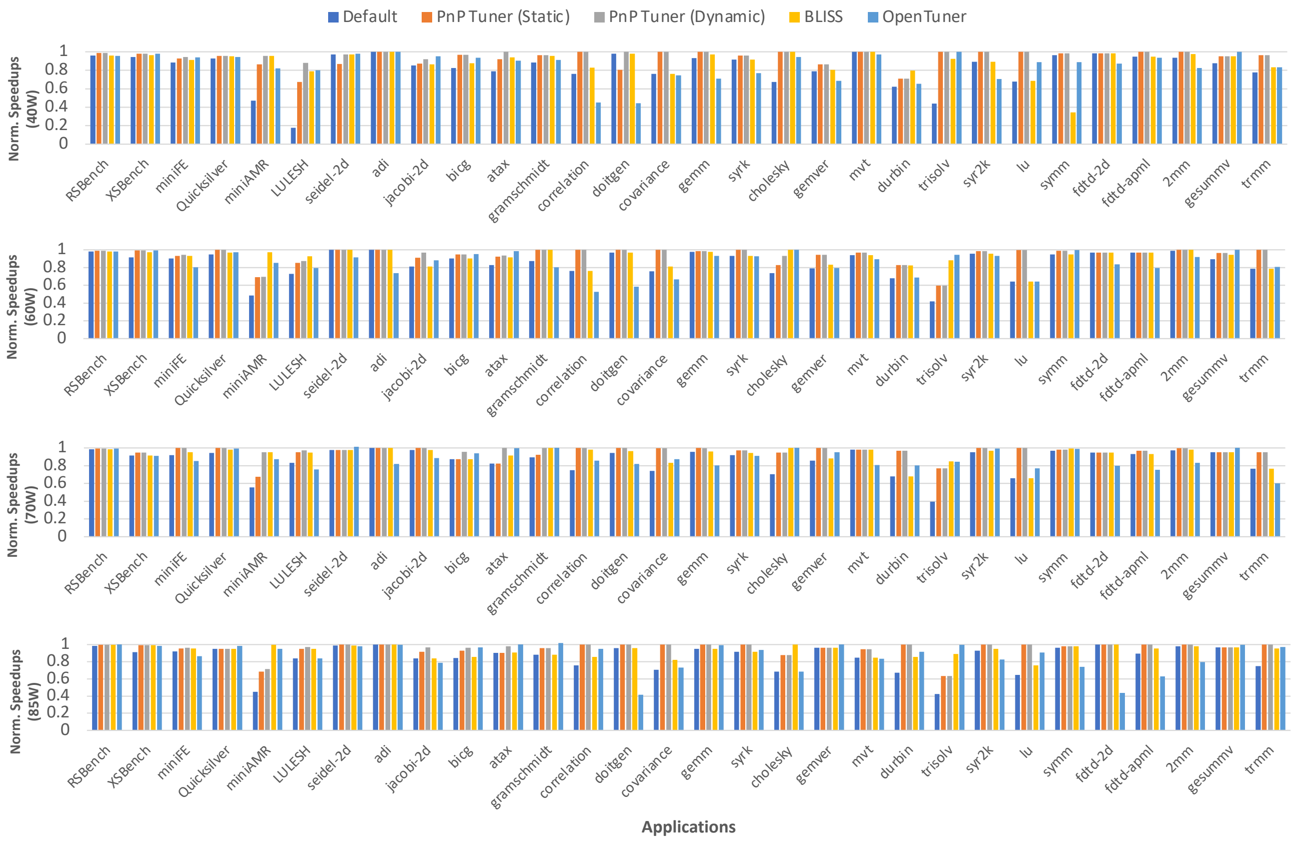


Fig. 3: Power Constrained Tuning (Haswell): Each chart shows results for a specific power constraint. Each bar-group shows geometric mean speedup for all OpenMP regions in an application over default OpenMP settings wrt the corresponding tuning approach. Speedups are normalized by oracle(brute-force) speedups. Normalized oracle speedups are always $1.0\times$. The PnP tuner outperforms BLISS in 82.5% and OpenTuner in 78% cases across all power constraints(see Section IV-B for details).

## IV. Experiments

To identify near optimum values of tuning parameters for both our experimental scenarios, we first explore every permutation of inputs and configurations considered in this study. We use this exhaustive exploration as an oracle to compare the results from our work. We also compare our work against BLISS [5] and OpenTuner [4]. All results presented in the following paragraphs represent speedups/greenups of each code region. For applications with multiple OpenMP regions, the geometric mean of speedups/greenups of all regions in an application are reported. We have also verified that there are sequences of serial code in between successive OpenMP regions. This allows us to look at each region as a self-contained unit, and makes them good candidates for tuning. We assume that the performance of these intervening serial sequences will not change and improving the performance of each OpenMP region would translate to improvement in application performance.

### A. Experimental Setup

For our experiments, we use two systems; one with Intel(R) Xeon(R) Gold 6142 CPU with 32 cores, two hyper-threads per core, and two sockets (Skylake) with a minimum and TDP package power of $75W$ and $150W$, and an Intel(R) Xeon(R) E5-2630 v3 CPU (Haswell), with 16 cores, two hyper-threads per core, and two sockets, and minimum and TDP package power of $40W$ and $85W$. We use Clang tools for code compilation and transformation to IR, and PyTorch DL libraries for building our GNN models.

### B. Power Constrained Auto-tuning

In this section, we evaluate the performance of our tuner in determining the optimal configuration for minimizing execution time given a specific power constraint (described in Section III-D2). To validate the effectiveness of our approach, we use *leave-one-out cross-validation (LOOCV)*. For each fold, code regions from one benchmark application is selected and assigned to the validation set and the code regions from all other applications are assigned to the training set. We repeat this process for all applications in our approach. Such a process is essential to evaluate the performance of our model on previously unobserved code regions.

The results for the Haswell system are shown in Figure 3. For each application, we calculate the geometric mean

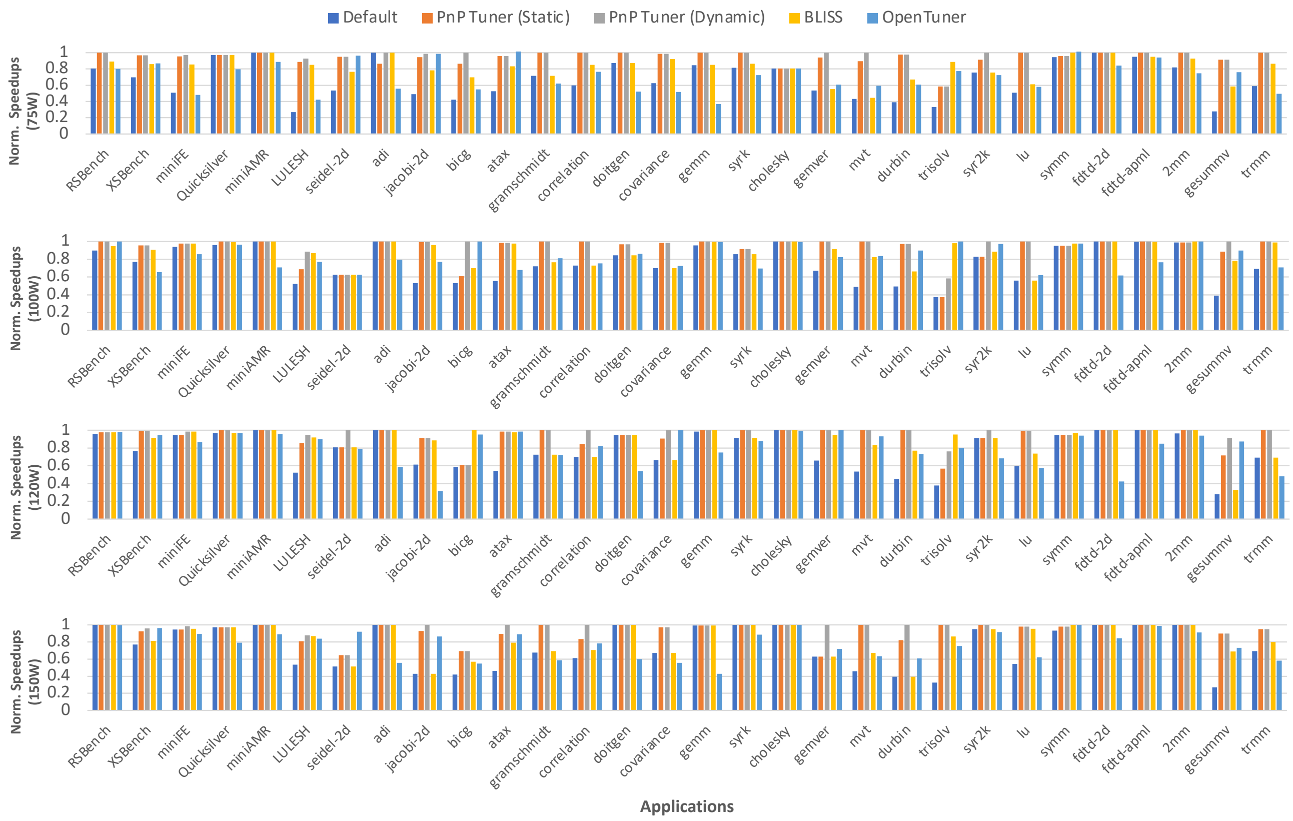


Fig. 4: Power Constrained Tuning (Skylake): Each chart shows results for a specific power constraint. Each bar-group shows geometric mean speedup for all OpenMP regions in an application over default OpenMP settings wrt the corresponding tuning approach. Speedups are normalized by oracle speedups. Normalized oracle speedups are always $1.0\times$. The PnP tuner outperforms BLISS in 85% and OpenTuner in 83% cases across all power constraints(see Section IV-B for more details).

speedups for all OpenMP regions in each application achieved by each tuner across four power constraints (i.e., 40W, 60W, 70W, 85W).

While training the model on the data from the Skylake system, we borrow ideas from transfer/inductive learning and perform an optimization step to speed up the training process. Because the code graphs are statically generated, the code graphs obtained on different systems using the same compiler are identical. For this reason, we save the weights and model states of the GNN model obtained while training our model on the Haswell system. While training the model on the Skylake data, we load the saved weights and model and only re-train the dense layers. This leads to $4.18\times$ faster training (or reduces training time by 76%).

Results for each power constraint (75W, 100W, 120W, 150W) is shown in Figure 4 for the Skylake system. Each speedup is normalized by the speedup achieved by the oracle (i.e., exhaustive exploration). In 74% cases (across both systems and power constraints), our PnP tuner identifies configurations that lead to $>= 0.95\times$ of the oracle speedups (assuming oracle as $1.0\times$). These results are obtained without executing the code. In contrast, BLISS and OpenTuner needs to execute code multiple times and achieves $>= 0.95\times$ of the oracle speedups in 51% and 34% cases respectively. The PnP tuner produces better results than BLISS and OpenTuner in 83% and 78% cases. Overall, the configurations predicted by our model lead to geometric mean speedups of $1.19\times$, $1.12\times$, $1.13\times$, and $1.14\times$ for power limits $40W$, $60W$, $70W$, and $85W$ on the Haswell system. In contrast, BLISS leads to speedups of $1.11\times$, $1.09\times$, $1.09\times$, and $1.11\times$ across these power constraints respectively. OpenTuner produces corresponding speedups of $1.06\times$, $1.0\times$, $1.04\times$, and $1.02\times$. On Skylake, our approach achieves geometric mean speedups of $1.5\times$, $1.25\times$, $1.26\times$, and $1.34\times$ across power constraints $75W$, $100W$, $120W$, and $150W$ respectively, compared to speedups of $1.29\times$, $1.2\times$, $1.18\times$, and $1.17\times$ produced by BLISS, and speedups of $1.27\times$, $1.13\times$, $1.07\times$, and $1.1\times$ produced by OpenTuner.

***Can performance counters further improve results?*** Although our approach leads to $>= 0.95\times$ of the oracle speedups in most cases, in approximately 8% of cases, our approach produces results which are $< 0.8\times$ of the oracle speedups. Previous works such as [43], [44] have used performance counters for tuning tasks. We borrow from these ideas to see if the results from our approach can be improved by using these as features (dynamic features). For this experiment, we update our model definition. We make no changes to the GNN layers. We repurpose the fully connected layers to accept as inputs five performance counters along with the ouputs from the GNN layers. We use PAPI [45] to collect counters related to L1, L2, L3 cache misses, number of instructions, and the number of mispredicted branches for each `OpenMP` region. These were selected as these have direct impact on code execution and performance.

We perform the same experiments as outlined in the previous paragraphs. However, we only validate on those applications whose speedups are $< 0.95\times$ on the oracle speedups. We see that by including performance counters, this approach identifies configurations that lead to $>= 0.95\times$ in $87.5\%$ cases (up from $74\%$). We show these results and comparisons in Figures 3 and 4. Therefore, a case can definitely be made for including performance counters for DL-based performance tuning. However, this comes at the additional cost of profiling. Profiling is necessary for generating the dataset to train the model. However, during inference, this approach (using both static and dynamic features) only needs to execute applications twice (to collect counters which serve as inputs to the model), which is less than other execution based tuners. To conclude, although this produces better results, it adds a profiling overhead. But during inference, in spite of this overhead it only needs two executions.

***Can we extend this approach to unknown power constraints?*** There might be scenarios where adding/removing new nodes to/from clusters, or other factors, might necessitate changing power constraints on nodes. Thus, our approach should also be generalizable to power constraints that our model has not been trained on, since data center policy changes may result in different power constraints being applied. To evaluate this scenario, we conduct four tests - two tests for each system - one test each for the lowest and highest power constraints considered in this paper. For each test, we first *exclude* all measurements for the target power constraint (e.g., for the 150W test on Skylake, for training, we use measurements from 75W, 100W, and 120W only). We then train and validate our model using *leave-one-out cross-validation* as before. This allows us to generalize for both unseen applications *and* unseen power constraints. However, unlike the initial experiments which uses a *static-only* approach, we use performance counters as part of the feature set in this experiment. This is to account for the variation in runtime behavior of parallel regions under varying power constraints. Static features cannot encapsulate such divergence in behavior. The input features and model is similar to the one described in Section IV-B. In addition to these features, we also input as feature the normalized power constraints for each feature set. This helps to associate runtime behavior (performance counters) with power limits.

Figures 5 and 6 shows that our model performs well in such scenarios for both the Skylake and Haswell systems, predicting configurations that are within $5\%$ (i.e., $\geq 0.95$ normalized speedup) of the best possible speedup in $64\%$ cases and within

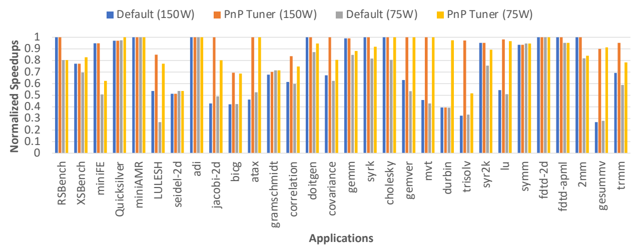


Fig. 5: Power Constrained Tuning on unseen power constraints (Skylake): Geometric mean speedup over default `OpenMP` settings. Results normalized by the oracle speedup.

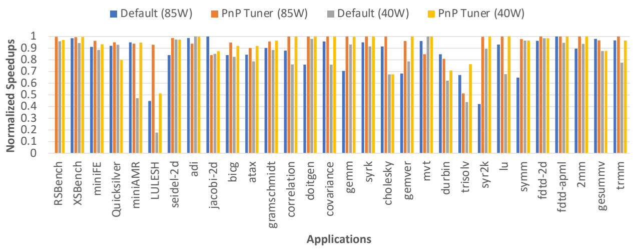


Fig. 6: Power Constrained Tuning on unseen power constraints (Haswell): Geometric mean speedup over default `OpenMP` settings. Results normalized by the oracle speedup.

$20\%$ of the best possible speedups in $85\%$ cases across both systems and four power constraints. On the Skylake systems, these tuning efforts lead to geometric mean speedups of $1.29\times$ and $1.36\times$ versus oracle speedups of $1.44\times$ and $1.59\times$ for power constraints of 150W and 75W respectively. On the Haswell system, these experiments produce speedups of $1.13\times$ and $1.17\times$ compared to oracle speedups of $1.16\times$ and $1.27\times$ for power constraints of 85W and 40W respectively.

TABLE II: Deep Learning Model Hyperparameters.

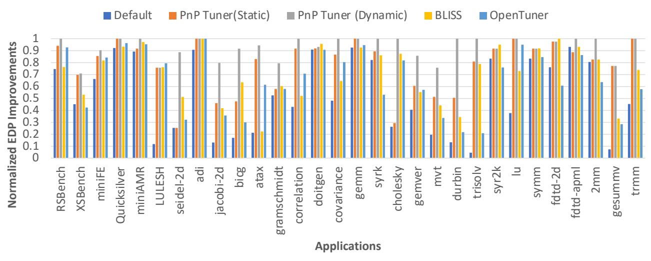
| Hyperparameter | Hyperparameter Values |
|---|---|
| Layers | RGCN (4), FCNN (3) |
| Activ. func. | Leaky ReLU, ReLU |
| Optimizer | AdamW (amsgrad) (Sec IV-B), Adam (Sec IV-C) |
| Learning Rate | 0.001 |
| Batch Size | 16 |
| Loss function | Cross Entropy Loss |

The hyperparameters of the models used in these experiments are shown in Table II. Other parameter values may have minor differences between experiments.

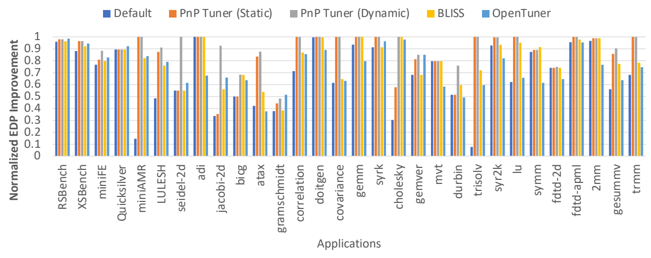
## C. Power and Performance Tuning

With increasing financial and environmental impacts of high energy usage, energy efficiency is now as important as performance in the current HPC landscape. However, simply optimizing for energy consumption, as shown in Section I, may lead to lower performance.

Thus, in this section, we outline the second scenario mentioned at the beginning of this section. To this end, we build



(a) Skylake



(b) Haswell

Fig. 7: Improvement in EDP over default `OpenMP` configurations for each application for the Haswell system. EDP improvements normalized in terms of best achievable EDP improvement.

a GNN-based tuner that uses only static features, with the aim of identifying a combination of power constraints and `OpenMP` runtime configurations that can lead to performance improvement while reducing energy consumption. As in the previous experiments, we model our flow-aware code graphs using an RGCN network. The outputs from the GNN layers are fed into the dense layers. These layers are trained with the target of finding configurations that produce the best energy-delay product (EDP). Again, we use *leave-one-out cross validation* to validate our model, and the process of assigning benchmark applications to the training and validation set is similar to that described in Section IV-B.

The configurations predicted in these experiments lead to within 5% of the oracle EDP improvements in 45% cases, and within 20% of the oracle improvements in 69% cases across the two systems. In comparison, `BLISS` reaches these numbers in 35% and 45% cases (Figure 7). `OpenTuner` reaches these numbers in 22% and 40% cases. Overall, the configurations predicted by our *static-only* approach leads to geometric mean improvements of 1.37× and 1.85× on the Haswell and Skylake systems compared to 1.31× and 1.69× respectively achieved by `BLISS` and 1.21× and 1.49× achieved by `OpenTuner`.

We have also analyzed the impact on execution time performance and energy consumption individually. Figure 8 shows the impact of tuning for EDP on execution time for both the Skylake and Haswell systems. Tuning for EDP leads to performance (time) improvement in 84% cases, and leads to slower execution than default settings in around 16% cases across both systems. On Skylake, all slowdowns are within 20% of the corresponding execution with all threads, while the geometric mean of all slowdows are within 14% of the default executions. On the Haswell system, there are fewer slowdowns, but the slowdowns are more significant with the largest slowdown within 30% of the default all-threaded execution, with the geometric mean within 23% of the default settings. Overall, excluding the cases that lead to slowdowns, tuning for EDP leads to 1.16× and 1.3× speedups on the Haswell and Skylake. In comparison, `BLISS` and `OpenTuner` leads to slowdowns in 28% and 46% cases respectively, with the largest slowdowns within 17% and 15% for `BLISS` and within 30% and 22% for `OpenTuner` on Haswell and Skylake.

We also show in Figure 8 the impact of tuning for EDP on energy. Across both systems, our approach predicts configurations that lead to reduction in energy consumption in 94% cases. In the remaining 6% cases, it predicts configurations that consume more energy than the default setting. However, the increase is minimal. On the Haswell, there is a 3% geometric mean increase in energy usage for those predictions. On the Skylake, the corresponding number is 1%. For the predictions that do lead to reduction in energy usage, there is a geometric mean greenup of 1.25× and 1.42× on the Haswell and Skylake respectively. In comparison, 2% of the predictions made by `BLISS` lead to increase in energy consumption. But, the overall greenups are slightly worse than the PnP Tuner (1.24× on the Haswell and 1.39× on the Skylake). The predictions made by `OpenTuner` lead to increase in energy consumption in 20% cases with overall greenups at 1.25× and 1.29× on the Haswell and Skylake respectively.

Similar to the experiment in Section IV-B, we also evaluate the effect of performance counters on EDP. As shown in Figure 7, adding performance counters to the feature set leads to improved results (predictions where the EDP is within 5% of the oracle moves up to 57% from 45% across both systems). Using performance counters leads to 77% cases where there is improvement in execution speed (down from 84%). This dichotomous behavior is the result of using a fused metric; because it is a product of both time and energy, the PnP tuner aims to tune for the best EDP. It might lead to scenarios where the reduction in energy might compensate for the increase in time. In this experiment, using performance counters leads to 95% cases where there are improvements in energy consumption. Overall, by using performance counters, the EDP predictions improve from 1.37× to 1.52× on the Haswell system, and from 1.85× to 2.31× on the Skylake. This lead to overall speedups of 1.13× and 1.39× on the Haswell and Skylake and greenups of 1.35× and 1.60× on the Haswell and Skylake systems.

## V. Discussion

Through this study, we have outlined a unique approach to two important problems in the HPC community. We have proposed a mechanism of tuning `OpenMP` configurations on power constrained systems. This is beneficial to data centers

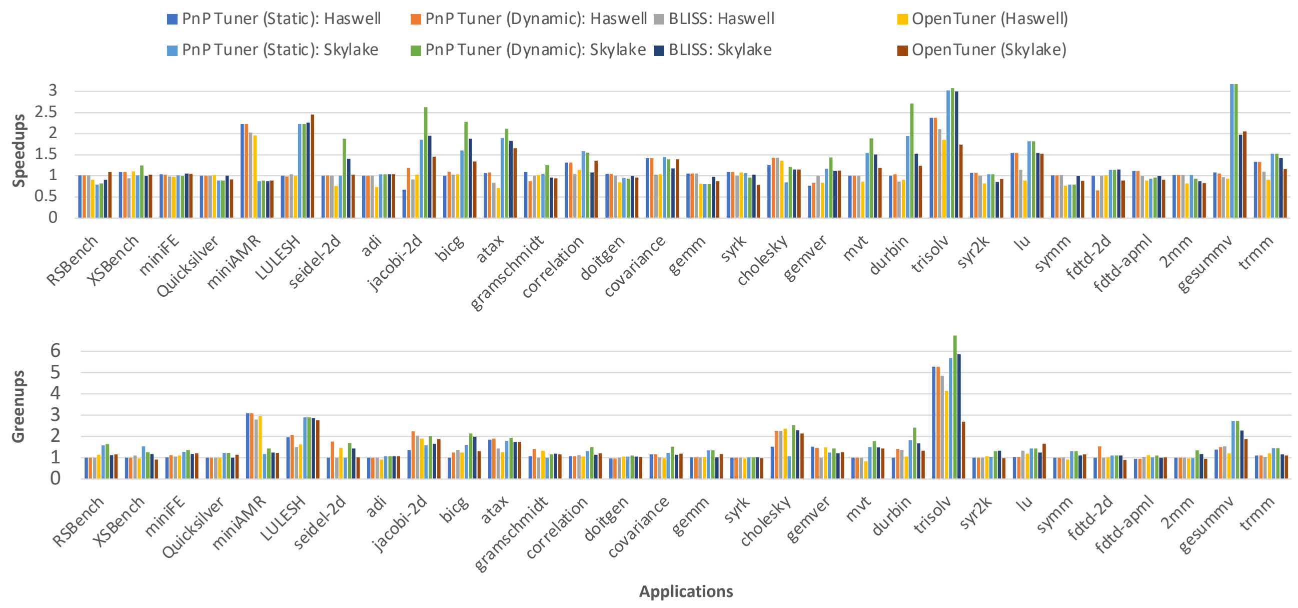


Fig. 8: Speedups/Greenups over default `OpenMP` configurations at TDP. Configurations are predicted to optimize for EDP.

and systems working under strict power budgets. As shown in previous sections, it is possible to considerably improve performance in such scenarios using our approach. Additionally, we also describe a method of identifying `OpenMP` configurations and power constraints that can lead to reduction in energy consumption with limited impact on execution time . To the best of our knowledge, this is the first work that aims to use GNN based techniques for these purposes.

As with all deep learning techniques, model training for several target systems/experiments might be burdensome. However, by using transfer learning techniques, we have reduced the training time on other systems by around 76% on a dataset of similar size (explained in Section IV-B). These optimizations can enable faster and easier deployment of such approaches on multiple systems.

Additionally, being a static approach, our tuner requires no sampling executions. This is in contrast to other tuners that need several sampling runs. Limiting these sampling runs, or setting a time-bound on the sampling phase to a small value leads to less than optimal results. Moreover, our approach was able to successfully identify most edge cases. For example, the `OpenMP` region in `trisolv` has the fastest execution with 1 thread in all cases. This is an outlier. Our approach could identify near optimal configurations in these cases as well with no executions.

## VI. Conclusion

In this work, we have outlined a twofold approach towards tuning `OpenMP` configurations in power constrained systems, as well tuning both `OpenMP` configurations and power constraints for execution time and energy consumption gains. We have used GNNs to model flow-aware code graphs to model the semantic and structural features of code regions. Our experiments show that the PnP Tuner can identify configurations that lead to improvements in execution time and energy consumption. In future, we aim to analyze the scalability of our approach to heterogeneous platforms and handheld devices.

## VII. Acknowledgements

This research was supported by the National Science Foundation under Grant number 2211982. We would also like to thank the ResearchIT team (*https://researchit.las.iastate.edu*) at Iowa State University for their constant support.

## References

[1] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. IEEE, 2010, pp. 189–194.

[2] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "Exploring hardware overprovisioning in power-constrained, high performance computing," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. New York, NY, USA: Association for Computing Machinery, 2013, p. 173–182.

[3] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: Towards automated performance tuning," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 2002, pp. 44–44.

[4] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[5] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1280–1295.

[6] M. A. S. Bari, N. Chaimov, A. M. Malik, K. A. Huck, B. Chapman, A. D. Malony, and O. Sarood, "Arcs: Adaptive runtime configuration selection for power-constrained openmp applications," in *2016 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 2016.

[7] M. A. S. Bari, A. M. Malik, A. Qawasmeh, and B. Chapman, "Performance and energy impact of openmp runtime configurations on power constrained systems," *Sustainable Computing: Informatics and Systems*, vol. 23, pp. 1–12, 2019.

[8] I. Karlin, J. Keasler, and J. R. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[9] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 661–672.

[10] M. A. Awan and S. M. Petters, "Race-to-halt energy saving strategies," *Journal of Systems Architecture*, vol. 60, no. 10, pp. 796–815, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762114001295

[11] J. H. Laros III, K. Pedretti, S. M. Kelly, W. Shu, K. Ferreira, J. Vandyke, and C. Vaughan, "Energy delay product," in *Energy-Efficient High Performance Computing*. Springer, 2013, pp. 51–55.

[12] P. Balaprakash, R. Egele, and P. Hovland, "ytopt," https://github.com/ytopt-team/ytopt, 2022.

[13] H. Menon, A. Bhatele, and T. Gamblin, "Auto-tuning parameter choices in hpc applications using bayesian optimization," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 831–840.

[14] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[15] 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)*. IEEE, 2022, pp. 391–400.

[16] A. Tehranijamsaz, M. Popov, A. Dutta, E. Saillard, and A. Jannesari, "Learning intermediate representations using graph neural networks for numa and prefetchers optimization," in *IPDPS 2022-36th IEEE International Parallel & Distributed Processing Symposium*, 2022.

[17] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica, "Neurovectorizer: End-to-end vectorization with deep reinforcement learning," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 242–255.

[18] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "Mlgo: a machine learning guided compiler optimizations framework," *arXiv preprint arXiv:2101.04808*, 2021.

[19] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. De Supinski, and M. Schulz, "Prediction models for multi-dimensional power-performance optimization on many cores," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 250–259.

[20] D. Li, B. R. de Supinski, M. Schulz, K. Cameron, and D. S. Nikolopoulos, "Hybrid mpi/openmp power-aware computing," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–12.

[21] C. C. De Oliveira, A. F. Lorenzon, and A. C. S. Beck, "Automatic tuning tlp and dvfs for edp with a non-intrusive genetic algorithm framework," in *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*. IEEE, 2018, pp. 146–153.

[22] W. Wang, A. Porterfield, J. Cavazos, and S. Bhalachandra, "Using per-loop cpu clock modulation for energy efficiency in openmp applications," in *2015 44th International Conference on Parallel Processing*. IEEE, 2015, pp. 629–638.

[23] A. Nandamuri, A. M. Malik, A. Qawasmeh, and B. M. Chapman, "Power and energy footprint of openmp programs using openmp runtime api," in *Energy Efficient Supercomputing Workshop*. IEEE, 2014, pp. 79–88.

[24] J. V. Ferreira Lima, I. Raïs, L. Lefevre, and T. Gautier, "Performance and energy analysis of openmp runtime systems with dense linear algebra algorithms," *The International Journal of High Performance Computing Applications*, vol. 33, no. 3, pp. 431–443, 2019.

[25] B. Rountree, D. H. Ahn, B. R. De Supinski, D. K. Lowenthal, and M. Schulz, "Beyond dvfs: A first look at performance under a hardware-enforced power bound," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2012, pp. 947–953.

[26] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. De Supinski, "Exploring hardware overprovisioning in power-constrained, high performance computing," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013.

[27] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[28] 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.

[29] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon, "Compiler-based graph representations for deep learning models of code," in *Proceedings of the 29th International Conference on Compiler Construction*, 2020.

[30] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[31] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European semantic web conference*. Springer, 2018, pp. 593–607.

[32] A. Dutta, J. Alcaraz, A. TehraniJamsaz, A. Sikora, E. Cesar, and A. Jannesari, "Pattern-based autotuning of openmp loops using graph neural networks," in *2022 IEEE/ACM International Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)*, 2022, pp. 26–31.

[33] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[34] W. Ma, M. Zhao, E. Soremekun, Q. Hu, J. M. Zhang, M. Papadakis, M. Cordy, X. Xie, and Y. L. Traon, "Graphcode2vec: generic code embedding via lexical and program dependence analyses," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 524–536.

[35] S. Brink, A. Marathe, T. Patki, and B. Rountree, "variorum," https://github.com/LLNL/variorum, 2022.

[36] L.-N. Pouchet *et al.*, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, vol. 437, pp. 1–1, 2012.

[37] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[38] J. R. Tramm, A. R. Siegel, B. Forget, and C. Josey, "Performance analysis of a reduced data movement algorithm for neutron cross section data in monte carlo simulations," in *International Conference on Exascale Applications and Software*. Springer, 2014, pp. 39–56.

[39] S. Hammond, C. Trott, and N. Evans, "minife," *GitHub repository*, 2022.

[40] A. Sasidharan and M. Snir, "Miniamr-a miniapp for adaptive mesh refinement," 2016.

[41] L. L. N. Lab, "Quicksilver," https://github.com/LLNL/Quicksilver, 2022.

[42] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[43] J. Alcaraz, A. TehraniJamsaz, A. Dutta, A. Sikora, A. Jannesari, J. Sorribes, and E. Cesar, "Predicting number of threads using balanced datasets for openmp regions," *Computing*, pp. 1–19, 2022.

[44] I. Sánchez Barrera, D. Black-Schaffer, M. Casas, M. Moretó, A. Stupnikova, and M. Popov, "Modeling and optimizing numa effects and prefetching with machine learning," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–13.

[45] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710. Citeseer, 1999.