Autotuning Energy-Delay Product 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 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 evaluate our approach on 30 benchmarks and proxy-/mini-applications with 68 `OpenMP` code regions. Our approach identifies `OpenMP` configurations for energy-delay product, that lead to performance improvement of 21% and energy reduction of 29% over the default `OpenMP` configuration at Thermal Design Power for a 32-core Skylake processor.**

***Index Terms*—Auto-tuning, OpenMP, GNN, EDP**

I. Poster Summary

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). 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, such as Intel's Running Average Power Limit (RAPL) currently exist for controlling power. However, a fixed power budget can *slow 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. 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.

`OpenMP`, as the de-facto parallel programming model for intra-node parallelism, provides a number of tunable parameters that highly influence code execution. This makes such a problem ideal for autotuning tasks. Auto-tuners have employed sophisticated techniques for tuning such search space. However, most tuners need multiple executions to identify profitable configurations, making them time consuming and resource intensive.

As a motivating example, consider the *ApplyAccelerationBoundaryConditionsForNodes* kernel from the `LULESH` 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 $7.54\times$, $2.11\times$, $1.80\times$ and $1.67\times$ over 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 = $\frac{Energy_{old}}{Energy_{new}}$) of $3.89\times$, but a speedup of $0.95\times$ (i.e., a *slowdown*). Therefore, optimizing for time and optimizing for energy may not yield the same `OpenMP` configuration. For applications where a slowdown is unacceptable, we can simultaneously optimize for time and energy by targeting the energy-delay product (EDP) metric [1]. To this end, we propose, PnP tuner, a graph neural network (GNN)-based technique that can be used to optimize for the *energy-delay product* to identify energy-efficient and performant configurations.

In this study, `OpenMP` code regions are first compiled to their intermediate representation (IR) and transformed

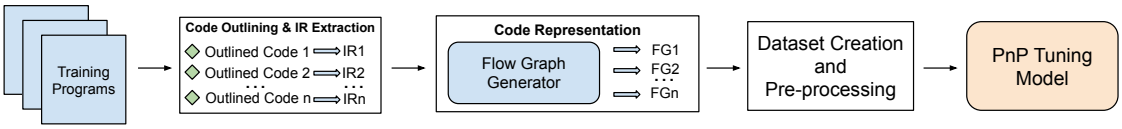

Fig. 1: PnP Tuner Pipeline: An overview of tasks in our GNN based power and performance tuner

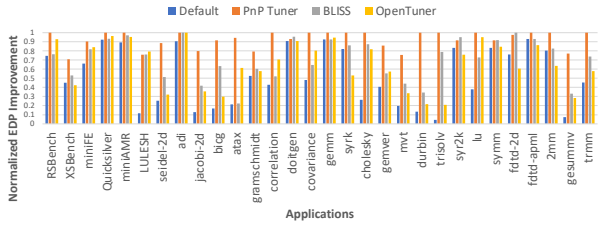

Fig. 2: Improvement in EDP over default `OpenMP` configurations for each application (normalized with respect to best EDP improvement for each application)

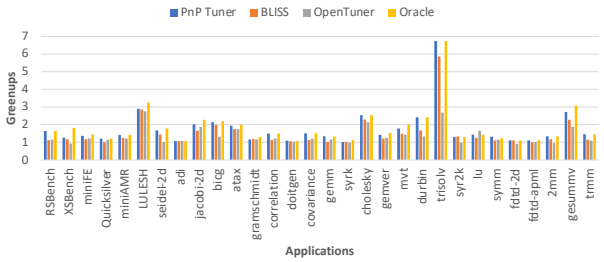

Fig. 3: Greenups over default `OpenMP` configurations at TDP. Configurations are predicted to optimize for EDP

to a flow-aware graph. Each of these graphs incorporates in them data, control, and call flow graphs. This allows us to model the semantics and structure of the input kernels. Each node in a graph contains an IR instruction. These form the node features in our GNN modeling. Edges between nodes contain information about the type of the flow (data, control, call) in addition to information about the source and target nodes. These form the edge features. We model each type of flow as a separate relation. Using a Relational Graph Convolutional Network (RGCN) aids us in modeling the three relations in our graphs. The modeled code features are then used for predicting the best configurations. In addition to the code graphs, we also use performance counters related to cache misses, instructions, and branch mispredictions. This allows the modeling of execution/runtime behavior of code kernels on target architectures.

TABLE I: Search space for tuning EDP.

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

To evaluate the effectiveness of our approach, we designed a search space as shown in Table I. Batches of graphs are fed into the GNN layers. The outputs

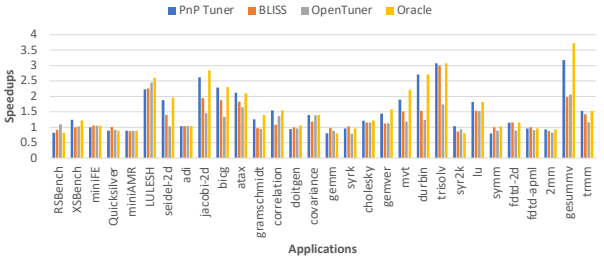

Fig. 4: Speedups over default `OpenMP` configurations at TDP. Configurations are predicted to optimize for EDP

from these layers are then concatenated with the pre-processed performance counters and are fed into the fully connected MLP layers. These MLP layers are used as the classifier to predict the best configuration. Our model uses AdamW as the optimizer and ReLU activations.

In this work, we have performed *leave-one-out-validation* to better show the results from each application considered in this study (Figure 2). We have also compared our results with two auto-tuners `BLISS` [2] and `OpenTuner` [3]. Our PnP tuner predictions lead to within 5% of the Oracle (brute-force best case) EDP predictions in 53% cases, and within 10% of Oracle predictions in 74% cases. The PnP tuner outperforms `BLISS` in 83% cases and `OpenTuner` in 97% cases.

We have also analyzed the speedups and greenups for the EDP configurations predicted by the PnP tuner (Figures 3 and 4). Our predictions lead to speedups in 67% cases and greenups in all cases. In terms of time, the PnP tuner outperforms `BLISS` and `OpenTuner` in 66% and 77% cases respectively. With respect to energy, the PnP tuner outperforms `BLISS` and `OpenTuner` in 90% and 93% cases respectively.

In future, we aim to adapt and improve this approach and apply it to similar tasks on GPUs.

References

[1] 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.

[2] 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.

[3] 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.