# DPGRAPH: Code Optimization with Dynamic Program Graphs using Heterogeneous GNNs

Quazi Ishtiaque Mahmud
*Iowa State University*
mahmud@iastate.edu

Lukas Rothenberger
*Technical University of Darmstadt*
lukas.rothenberger@tu-darmstadt.de

Le Chen
*Argonne National Laboratory*
le.chen@anl.gov

Felix Wolf
*Technical University of Darmstadt*
felix.wolf@tu-darmstadt.de

Ali Jannesari
*Iowa State University*
jannesar@iastate.edu

## I. INTRODUCTION

In this work, we propose DPGRAPH, a dynamic program graph representation for learning-based code performance optimization on heterogeneous systems that augments conventional LLVM intermediate representation (LLVM-IR) program graphs with runtime data-dependence information. Most existing LLVM-IR graph representations [1]–[4], [12] rely primarily on static program information. Although static graphs capture control, data, and call relationships, dynamic dependencies capture dynamically computed memory accesses. Consequently, static dependencies do not always reflect how a program actually behaves for a particular execution and input. DPGRAPH addresses this limitation by combining fine-grained LLVM-IR based program representation with runtime dependencies, creating an execution-flow-aware program representation that can be used to predict high-performance optimization configurations.

## II. APPROACH

**Incorporating Dynamic Dependencies.** The DPGRAPH pipeline begins by compiling source code into fine-grained LLVM-IR. We then perform hybrid program analysis using DiscoPoP [10], combining static analysis with dynamic profiling. During execution, DiscoPoP tracks memory accesses and identifies important dependence relationships, particularly read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependencies. These dependencies are associated with individual LLVM-IR instructions rather than only source-code lines. This instruction-level mapping provides substantially finer resolution where multiple LLVM instructions originating from the same source line can be distinguished and connected to their actual runtime dependencies.

The resulting static and dynamic dependencies are converted into triplets containing a source instruction, dependency type, and destination instruction. These dependencies are then inserted as additional typed edges into the LLVM-IR based program graph. The resulting DPGRAPH contains control-flow, data-flow, call-flow. The data flow is cateogorized into three data dependencies: RAW, WAR, and WAW. This representation therefore combines the compiler-visible structure of the program with dependencies observed during actual execution. Figure 1 shows the DPGRAPH creation process.

**Incorporating Input Size.** DPGRAPH also incorporates input size, because the best optimization configuration can change even when program structure remains identical. For example, considering the task of predicting optimal vectorization and interleave factor (VF and IF) prediction. The optimal value for VF/IF may vary for different input sizes. Thus, input size is explicitly incorporated into the DPGRAPH representation, enabling it to distinguish programs that look structurally similar but exhibit different performance behavior under different inputs.

**Graph Processing & Configuration Prediction.** The graph is processed using a Heterogeneous Graph Neural Network (HGNN). LLVM-IR instructions are tokenized and embedded, while different edge types are encoded using one-hot-encoding mechanism. GAT-based graph-convolution layers model relationships among the heterogeneous node and edge types. During HGNN training, Graph pooling produces a single representation for the entire program, which is passed through a linear transformation and Softmax layer to predict the optimal configuration.

## III. RESULTS

DPGRAPH is evaluated on three major compiler-optimization tasks: i) Vectorization and Interleave Factor (VF/IF) prediction, ii) CPU/GPU (OpenMP simd/target) based parallelism prediction & iii) Optimal Loop Order prediction. For all experiments oracle speedup is calculated by using brute force search to find the optimal configuration.

**VF and IF prediction.** We train (80% training, 10% validation and 10% testing) on 6.5k samples from the NeuroVectorizer dataset [11]. We compare DPGRAPH with both classical (Polly [6]) and ML-based tools. The considered IF configs (classes) are: 1, 2, 4, 8, and 16. The considered VF configs (classes) are: 1, 2, 4, 8, 16, 32, and 64. Figure 2 shows that DPGRAPH surpasses state-of-the-art MIREncoder

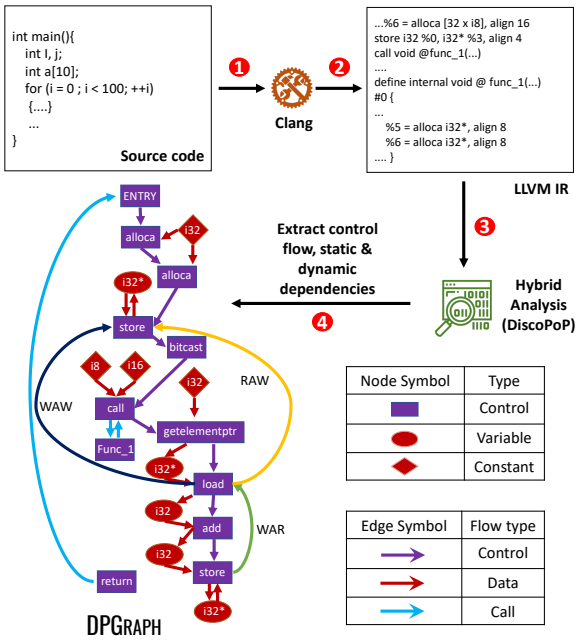



Fig. 1. DPGRAPH is constructed by combining control flow with static and dynamic data dependencies (WAW, RAW, WAR).

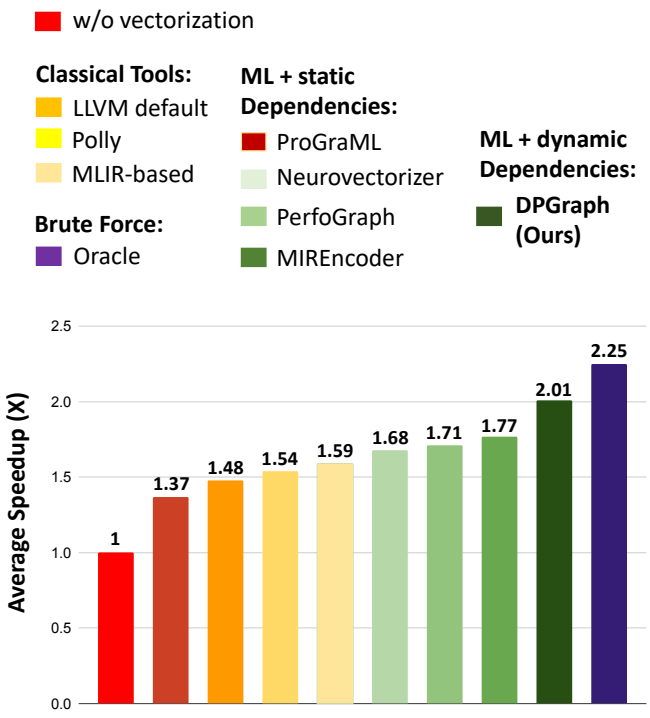



Fig. 2. Comparing DPGRAPH with state-of-the-art methods for VF/IF based speedup..

[12] by achieveing 13.6% (1.77x vs 2.01x) better speedup. The speedup is calculated as the ratio of program runtime using $< VF = 1, IF = 1 >$ value and the predicted $< VF, IF >$ value using DPGRAPH. In addition to that, for training DPGRAPH uses 40x fewer (6.5k vs 275k) training samples than MIREncoder which makes it more compute efficient.

**Simd and Target prediction.** We also employ DPGRAPH to predicts whether a loop should use OpenMP simd or GPU target parallelization. ConTraPh dataset [4] of 1.6k samples (80%, 10%, 10% split) is used which is derived from NPB [9], Rodinia [7] and Dataracebench [8]. Table I shows DPGRAPH achieves 15.61× speedup, compared with 14.58× for ConTraPh [4]. Importantly, DPGRAPH requires roughly one hour of model training plus six hours of profiling, whereas ConTraPh requires more than 52 hours of training, corresponding to approximately an 86% reduction in training time.

TABLE I
COMPARISON OF DPGRAPH WITH OTHER APPROACHES FOR CPU/GPU PARALLELIZATION. OS = ORACLE SPEEDUP.

| Model | Accuracy | Speedup (OS = 16.15x) |
|---|---|---|
| ProGraML [1] | 0.91 | 6.67x |
| PerfoGraph [2] | 0.92 | 9.82x |
| AutoParLLM [3] | 0.92 | 10.09x |
| ConTraPh [4] | 0.96 | 14.58x |
| **DPGRAPH** | **0.97** | **15.61x** |

**Optimal loop-order prediction**. For this task, we use the dataset proposed by [5]. The dataset contains 331 code snippets containing 2D kernels, where each kernel is transformed into four loops after tiling, resulting in the iteration space $(i, j, ii, jj)$. The dataset considers the following 6 legal loop orders with and without unrolling-and-jam operation (in total 12 different classes) for each 2D kernel: $(i, j, ii, jj), (i, j, jj, ii), (i, ii, j, jj), (j, i, ii, jj), (j, i, jj, ii)$, $(j, jj, i, ii)$. Table II shows that DPGRAPH achieves 7.05% (1.56x vs 1.67x) better speedup comparing to state-of-the-art method PerfoGraph [2].

TABLE II
COMPARING DPGRAPH WITH OTHER APPROACHES ON LOOP ORDER PREDICTION TASK. OS = 'ORACLE SPEEDUP'.

| Approach | Accuracy | Speedup (OS = 1.7x) |
|---|---|---|
| Polly [6] | 0.54 | 1.52x |
| PrograML [1] | 0.67 | 1.53x |
| UJOpt [5] | 0.70 | 1.54x |
| PerfoGraph [2] | 0.74 | 1.56x |
| **DPGRAPH** | **0.83** | **1.67x** |


## ACKNOWLEDGEMENT

This work is supported by NSF grants #2211982 and #2426580.


## REFERENCES


[1] Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., O'Boyle, M. F., & Leather, H. (2021, July). Programl: A graph-based program representation for data flow analysis and compiler optimizations. In International Conference on Machine Learning (pp. 2244-2253). PMLR.Soc. London, vol. A247, pp. 529–551, April 1955.

[2] TehraniJamsaz, A., Mahmud, Q. I., Chen, L., Ahmed, N. K., & Jannesari, A. (2023). Perfograph: A numerical aware program graph representation for performance optimization and program analysis. Advances in Neural Information Processing Systems, 36, 57783-57794.

[3] Mahmud, Q. I., TehraniJamsaz, A., Phan, H. D., Chen, L., Capotă, M., Willke, T. L., ... & Jannesari, A. (2025, April). AutoParLLM: GNN-guided context generation for zero-shot code parallelization using LLMs. In Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers) (pp. 11821-11841).

[4] Mahmud, Q. I., TehraniJamsaz, A., Ahmed, N. K., Willke, T. L., & Jannesari, A. (2025, June). Contraph: Contrastive learning for parallelization and performance optimization. In Proceedings of the 39th ACM International Conference on Supercomputing (pp. 596-610).

[5] Babalad, S., Shevade, S. K., Thazhuthaveetil, M. J., & Govindarajan, R. (2025, June). UJOpt: Heuristic Approach for Applying Unroll-and-Jam Optimization and Loop Order Selection. In Proceedings of the 39th ACM International Conference on Supercomputing (pp. 611-624).

[6] Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A., & Pouchet, L. N. (2011, April). Polly-Polyhedral optimization in LLVM. In Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT) (Vol. 2011, p. 1).

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

[8] Liao, C., Lin, P. H., Asplund, J., Schordan, M., & Karlin, I. (2017, November). DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-14).

[9] Jin, H. Q., Frumkin, M., & Yan, J. (1999). The OpenMP implementation of NAS parallel benchmarks and its performance.

[10] Li, Z., Atre, R., Ul-Huda, Z., Jannesari, A., & Wolf, F. (2015, August). DiscoPoP: A profiling tool to identify parallelization opportunities. In Tools for High Performance Computing 2014: Proceedings of the 8th International Workshop on Parallel Tools for High Performance Computing, October 2014, HLRS, Stuttgart, Germany (pp. 37-54). Cham: Springer International Publishing.

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

[12] Dutta, A., Jannesari, A. (2024, October). Mirencoder: Multi-modal ir-based pretrained embeddings for performance optimizations. In Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques (pp. 156-167).