Intelligent CUDA parameter tuning for triangle counting: A GCN approach to predicting runtime block sizes

Hasan Serdar Arikan*, Rakibul Hassan*, Shubhashish Kar*, Doru Thom Popovici†, Shaikh Arifuzzaman*
*University of Nevada, Las Vegas, Las Vegas, NV, USA †Lawrence Berkeley National Laboratory, Berkeley, CA, USA
{arikan,hassar2,kars1}@unlv.nevada.edu, dtpopovici@lbl.gov, shaikh.arifuzzaman@unlv.edu

***Abstract*—Determining optimal CUDA block size configurations represents a critical challenge in GPU-based graph processing. The block size directly impacts execution efficiency by balancing kernel launch overhead and GPU occupancy—with our experiments showing performance variations of up to 24x. Existing approaches rely on architectural constraints such as static API-based prediction methods that fail to account for the dynamic memory requirements of graph algorithms, which vary significantly based on graph topology. We propose a novel Graph Convolutional Network (GCN) approach that predicts optimal block sizes (64, 128, 256, 512, or 1024 threads) for CUDA graph algorithms using only the graph edge list as input. Our three-layer GCN model uses node degree as the primary feature, capturing the relationship between graph structure and execution performance. Trained on over 100 diverse networks from multiple domains (social, biological, web, road networks, etc.), our model effectively learns how topological characteristics influence memory access patterns and workload distribution. Our model predicts optimal CUDA block size with an accuracy of 63%. When the model fails to predict the optimal block size, it typically selects a suboptimal size that is adjacent to the optimal configuration. Experimental results demonstrate that our GCN-based model significantly outperforms fixed block size strategies, achieving up to 5.3× mean performance improvement across diverse graph types. This work bridges graph learning and high-performance computing, enabling automatic parameter tuning without exhaustive empirical search, and opens avenues for further optimization in graph algorithm acceleration.**

***Index Terms*—GPU computing, graph neural networks, triangle counting, CUDA, intelligent tuning**

I. Introduction

CUDA block size represents one of the most critical configuration parameters in GPU kernel optimization. Block size defines the number of threads that execute together within a single thread block. This parameter directly influences GPU performance through its impact on resource utilization, memory access patterns, and execution efficiency. When a CUDA kernel launches, threads within each block share local memory resources and execute on the same streaming multiprocessor, making block size selection fundamental to achieving optimal performance.

Block size selection fundamentally impacts GPU performance through its influence on resource allocation and execution efficiency. The choice involves balancing multiple competing factors: larger blocks efficiently distribute kernel launch costs across more threads but may limit the number of concurrent blocks that can execute on streaming multiprocessors, while smaller blocks enable higher occupancy and better resource utilization but incur proportionally higher launch overhead. This configuration also determines memory bandwidth utilization and the GPU's ability to mask memory latency through parallel execution. These interdependent factors create a complex optimization space where performance varies significantly with block size. Each algorithm-dataset combination possesses a unique optimal configuration that maximizes execution efficiency, making accurate block size selection critical for peak performance [1].

Current approaches to block size determination in CUDA programming rely on several API function calls [2]. The NVIDIA CUDA Toolkit provides occupancy calculators such as *cudaOccupancyMaxActiveBlocksPerMultiprocessor* that predict occupancy based on block size and shared memory requirements. Additionaly other occupancy based APIs such as *cudaOccupancyMaxPotentialBlockSize* and *cudaOccupancyMaxPotentialBlockSizeVariableSMem* heuristically calculate execution configurations to maximize multiprocessor-level occupancy. These API calls assist programmers in choosing the thread block size based on register and shared memory requirements. However, these static prediction methods have significant limitations, particularly for graph algorithms operating on sparse data structures. On graph algorithms, the need for shared memory can vary depending on runtime parameters that the API call can't predict just by analyzing the kernel code. The memory requirements for such algorithms can vary considerably at runtime depending on graph topology and characteristics [1].

This limitation becomes particularly evident when processing large-scale graphs. As shown in Figure 1, in triangle counting algorithm, massive network graphs such as the Web-indochina-2004 graph (7 million nodes and 194 million edges) show execution times ranging from 9 to 24 seconds. The performance gap widens further on larger networks. On the Friendster social network graph performance varies from 8 to 37.5 seconds depending on block size configuration. It clearly demonstrates that there is neither single block size nor specific pattern for the best performance in different large-scale graph. Additionally, determining the optimal block size becomes increasingly challenging for massive graphs with billions of edges and nodes, and dynamic graphs where topology continuously evolves. The existing tools provide insufficient guidance for these complex scenarios.

In this paper, we propose a machine learning approach using

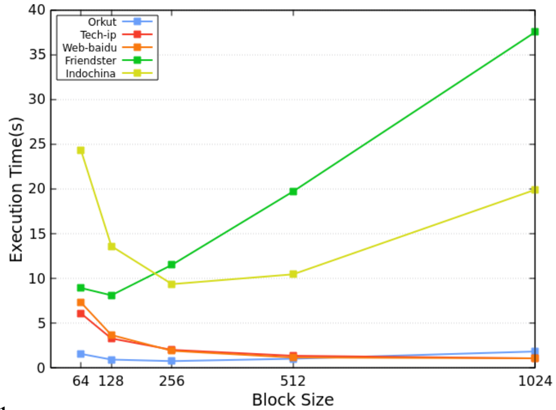

Fig. 1: Execution Time vs Cuda Block-size for Triangle Counting kernel. The plot demonstrates that **different graphs are experiencing different trends in performance with the increment of Cuda Block Size**.

Graph Convolutional Networks to predict optimal block size for CUDA graph algorithms. As shown in figure 2, we trained the model using diverse set of graph networks of various sizes and topologies. Our model requires only the edge list of a graph as input and directly incorporates graph properties into CUDA parameter optimization, enabling more targeted and effective tuning for graph processing on GPU architectures. Our key contributions are summarized as follows:

1) We introduce a **novel GCN-based framework** for CUDA block size prediction, designed specifically for graph algorithms. To the best of our knowledge, this is the **first** framework to leverage **graph topology** to predict CUDA runtime parameters, bridging the gap between graph learning and GPU optimization.
2) Our GCN achieves 63% accuracy in predicting the optimal CUDA block size with 5 different classes, and even its suboptimal predictions are often adjacent to the optimal value. Compared to fixed-size strategies, our approach improves execution time by up to **5.3**$\times$.
3) Unlike static occupancy-based tools that rely solely on kernel structure, our method operates directly on the graph's edge list and captures how graph structure impacts memory behavior and kernel performance.
4) We run extensive training on **100+ diverse real-world graphs**, spanning domains such as social networks, road maps, biological systems, and web graphs. This broad dataset enables our model to generalize across varied structural patterns.

II. Background

A. Triangle Counting Algorithm

Graph algorithms implemented on GPU architectures present unique optimization challenges due to their irregular memory access patterns and workload distributions[3]. Likewise, triangle counting[4] incorporates the challenges including total amount of work, workload imbalance among the workers, and memory access patterns limited by GPU memory bandwidth. Triangle is one of the basic motifs to get the insights of large-scale networks and it serves as a great case study for analyzing the impact of CUDA block size selection on performance[1].

We implemented the node iterator algorithm, which leverages the node ordering method to optimize memory usage and execution time. This approach strategically stores a node's neighbors in memory only when the neighbor's ID is less than the node itself [5]. By implementing this selective storage strategy, we significantly reduce memory requirements while maintaining algorithmic correctness. Our implementation follows a vertex-centric approach to triangle enumeration[6]. As shown in Algorithm 1, the algorithm systematically iterates through each node $v \in V$ and identifies the neighbors of v forming a wedge hinging at vertex v, where $v \prec u$ (id based comparison, v has a smaller ID than u) to avoid duplicate counting. While the wedge is found, the algorithm checks whether the wedge is closed or not by verifying $(u, w) \in E$. If the wedge is closed, they form a triangle in the graph structure, and the triangle count T is increased.

Algorithm 1 Vertex-centric Triangle Counting

for $v \in V$ **do**
  for $u \in N_v$ and $v \prec u$ **do**
    for $w \in N_v$ and $u \prec w$ **do**
      if $(u, w) \in E$ **then**
        $T \leftarrow T + 1$

We developed a parallel GPU implementation of this triangle counting algorithm to leverage massive parallelism. Each GPU thread processes a specific vertex, executing the nested loops from Algorithm 1 in parallel across hundreds of concurrent threads. This strategy distributes computational workload across multiple streaming multiprocessors, significantly reducing execution time compared to sequential implementations. Despite these performance gains, the irregular nature of graph structures creates several optimization challenges. Memory coalescing issues arise from unpredictable access patterns, while varying vertex degrees lead to workload imbalances across threads. These challenges make CUDA block size selection critical for optimal performance. For detailed implementation specifics, refer to [1].

III. Related Work

Selecting the optimal configuration in high-performance computing (HPC) systems from a vast search space is a challenging task. Factors such as algorithmic design, system architecture, and dataset characteristics significantly influence runtime performance. In [7], a suite of machine learning models is employed to predict suitable configurations, including hardware and resource allocation parameters, particularly in the context of genetic algorithms. The configurations recommended by the model lead to a 22% improvement in runtime performance compared to the baseline generic setup. In [8], Zhang et al. discuss the challenges associated with deep graph convolutional networks (GCNs), while also acknowledging their strengths in enabling improved representation learning. They specifically highlight the issue of over-smoothing in GCNs, which becomes more pronounced as the depth of the

Group	Subgroup	#	Vertices		Edges		Max Deg		Avg Deg		Triangle Count	
			Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
SNAP	Social/Comm.	7	4.0K	1.7M	27.8K	11.1M	172	35.5K	6.6	80.9	45K	54.1M
	Road	3	1.1M	2.0M	1.5M	2.8M	9	12	2.8	2.8	67K	121K
	Location	2	58.2K	196.6K	214.1K	950.3K	1.1K	14.7K	7.4	9.7	495K	2.3M
	Ground-Truth	5	317.1K	3.1M	925.9K	117.2M	343	33.3K	5.3	76.3	667K	628M
SS	Road	9	114.6K	50.9M	119.7K	54.1M	6	13	2.1	2.4	68	439K
NR	Biological	17	636	15.2K	1.1K	246.0K	37	913	2.3	60.5	29	2.2M
	Brain	8	65	734.6K	730	165.9M	45	15.1K	10.0	451.7	3.6K	20.8B
	Collaboration	10	11.2K	1.1M	91.3K	56.3M	279	11.5K	4.9	224.1	4.4K	4.9B
	Economic	4	497	15.6K	17.5K	410.8K	490	2.7K	2.2	261.6	1.1K	25.3M
	Graph500/Bench	6	4.0K	2.3M	3.8M	56.2M	28	170.5K	6.0	3712.6	16.9M	8.5B
	Heterogenious	5	100.1K	2.4M	739.8K	31.4M	9.4K	237.3K	10.9	32.2	819K	849.3M
	Recommmendation	8	61.0K	2.1M	439.8K	49.7M	3.7K	158.9K	5.4	726.9	80K	1.2B
	Sci-Computing	8	38.1K	8.4M	1.3M	25.2M	28	16.4K	6.0	845.5	878	190M
	Tech	8	2.1K	2.3M	6.6K	21.6M	95	1.8M	4.0	19.2	2.0K	2.3M
	Web	5	12.3K	2.1M	19.5K	17.0M	59	97.8K	3.2	21.4	10K	25.2M

Fig. 2: **Network Dataset Statistics** by Group. Note: K = thousands, M = millions, B = billions. SNAP = Stanford Network Analysis Project, SS = SuiteSparse, NR = Network Repository. Social/Comm = Social and Communication networks, Graph500/Bench = Graph500 and Benchmark (bhoslib) networks, Ground-Truth = Networks with ground truth communities.

network increases. In [9], they propose a framework to detect the optimal thread-block size in CUDA kernels depending on the warp-occupancy, the block-occupancy, and the grid-occupancy. In this study, they run experiments on memory-bound Basic Linear Algebra Subprograms. Connors. et al.[10], demonstrate that performance can vary significantly for thread structure instead of providing the highest occupancy. They propose a Support Vector Machine based model achieving 83.7% accuracy with 1.8× speedup compare to the baseline versions. Moreover, the study[11] leverages iterative machine learning based regression model to adjust GPU block size. The suite of regression techniques include KNN, SVM, Random Forest and etc. However, these existing approaches primarily target general-purpose CUDA kernels or focus on architectural constraints without considering the unique characteristics of graph algorithms.

Graph algorithms present distinct optimization challenges due to their irregular memory access patterns, topology-dependent workload distributions, and varying computational requirements based on graph structure. Our work is specifically designed to address these challenges by leveraging GCNs to capture the intrinsic relationship between graph topology and optimal CUDA execution parameters.

IV. Methodology and Implementations

A. Methodology Overview

We developed a comprehensive machine learning pipeline to address the relationship between graph topology and optimal CUDA performance parameters. Figure 3 presents our research pipeline from data collection to model deployment. Our methodology encompasses four main phases:

1) Data Collection: To ensure robust training and evaluation of our graph neural network model, we assembled a diverse collection of over 100 graph networks from three major repositories: the *SNAP Dataset*, the *SuiteSparse Matrix Collection*, and the *Network Data Repository*. As summarized in Table 2, these sources collectively contribute a wide variety of graph types, including *social networks*, *biological networks*, *web graphs*, *road networks*, *collaboration networks*, and others.This dataset was carefully curated to provide comprehensive coverage of graphs with diverse *sizes*, *densities*, and *topological structures*—key factors that significantly influence CUDA kernel behavior and performance. A broad range of networks enables our GCN model to learn generalized patterns that map graph features to optimal runtime configurations.

2) Graph Preprocessing: standardizes the collected graphs through zero-indexing normalization, as some datasets use non-zero starting node IDs. This preprocessing step ensures consistent input format for subsequent performance measurement and eliminates potential indexing conflicts during algorithm execution.

3) Performance Measurement: executes our Kokkos-based triangle counting implementation across five different CUDA block sizes (64, 128, 256, 512, 1024 threads) for each graph. We perform ten independent runs for each dataset and compute the mean execution time of each block size to ensure reliable performance measurements. This phase generates comprehensive timing data that serves as ground truth for optimal block size determination, capturing the performance characteristics of each graph-block size combination across diverse dataset.

4) Machine Learning Pipeline: encompasses a graph neural network framework that learns to map graph structural properties to optimal CUDA block sizes. It includes feature engineering using node degrees, one-hot encoding of optimal block sizes as target labels, GCN model training with three convolutional layers, and implementation of neighbor sampling and mini-batch strategies for scalable training across graphs of varying sizes.

The subsequent sections detail the specific implementation aspects of our GCN-based approach.

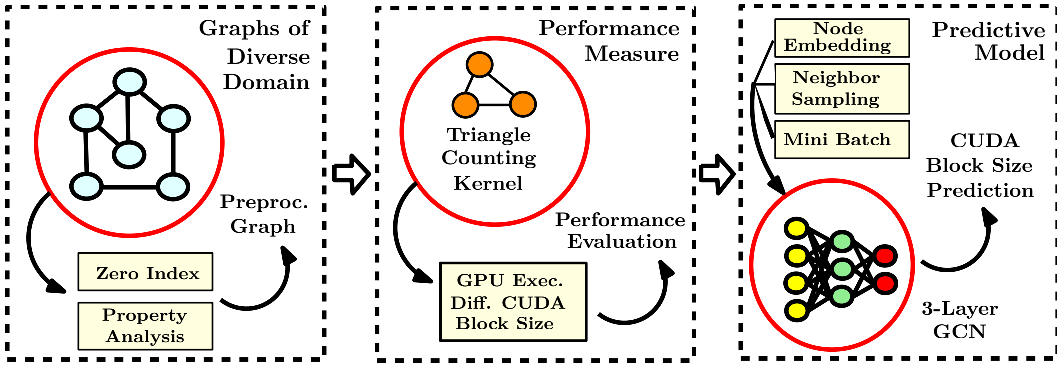

Fig. 3: **Methodology Overview:** End-to-end pipeline detailing the process from data acquisition and preprocessing to runtime performance profiling and GCN-based training for predicting optimal CUDA block sizes.

B. Graph Convolutional Networks

In order to predict optimal CUDA block size for graph algorithms, we developed a Graph Convolutional Network (GCN) model that learns the relationship between graph structure and execution performance. Our model architecture was designed to capture the topological characteristics of graphs that influence memory access patterns and workload distribution during parallel execution [12] [13] [14].

The selection of GCN as our architecture was motivated by several key factors: The ability of GCNs to capture localized structural patterns within graph data [14], the computational efficiency of graph convolutions compared to traditional approaches that require manual feature engineering to extract graph-level statistics (such as degree distributions, clustering coefficients, or graph invariants) for use in standard neural networks, and the proven effectiveness of GCNs in graph-level classification tasks across diverse domains [12].

These properties make GCNs particularly well-suited for our task, as the optimal block size is determined by how graph structure influences memory access patterns. The core of our model consists of three message passing layers to capture the topological characteristics of graphs that influence memory access patterns during CUDA kernel execution. Our implementation employs three GCN convolutional layers design which allows each node to collect and process features from nodes up to three hops away. This depth provides sufficient structural context for the model to understand local graph patterns while maintaining computational efficiency. The three-layer architecture represents a balance between capturing meaningful neighborhood information and avoiding the over-smoothing problem common in deeper GCN architectures, where excessive layers can cause node representations to become overly similar and lose discriminative power. Then, global mean pooling to aggregate node-level features into a graph-level representation, and a final MLP classifier to predict the optimal block size.

C. Feature Selection

We conducted initial experimentation with multiple graph properties including node degree, clustering coefficient and PageRank scores to determine the most effective node features for our message-passing implementation. Our preliminary analysis revealed that while clustering coefficient provides valuable structural information about local connectivity patterns, its computational complexity makes it challenging to process large-scale graphs with millions of nodes and high-degree hubs. In contrast, node degree computation requires only O(1) time per node (as NetworkX stores the neighbor count of each node in a dictinoary) [15] and captures the essential characteristics needed for CUDA block size prediction. Our empirical evaluation demonstrated that using degree alone achieved a promising predictive performance while maintaining computational efficiency. Consequently, we focused on a single but highly informative characteristic for node features.

D. Architecture Details & Training Objective

Our GCN architecture, illustrated in Figure 4, processes graph structure through four distinct stages to predict optimal CUDA block sizes. The model begins with graph input featuring node degree as the primary feature, followed by three GCN layers for message passing, global pooling for graph-level representation, and finally an MLP classifier for block size prediction.

Each message passing layer aggregates information from neighboring nodes using the GCNConv operation. The key symbols used in mathematical formula are as follows: A represents the adjacency matrix of the input graph, $H^{(l)}$ denotes the node feature matrix at layer l, $W^{(l)}$ is the trainable weight matrix, and σ represents the activation function. The GCN update rule follows:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{1}$$

where $\tilde{A} = A + I$ is the adjacency matrix with self-loops, $\tilde{D}$ is the diagonal degree matrix of $\tilde{A}$, $H^{(l)}$ is the node feature matrix at layer l, $W^{(l)}$ is the trainable weight matrix, and σ is the ReLU activation function.

After the message passing phase, we employ global mean pooling to obtain a graph-level embedding:

$$h_G = \frac{1}{|V|}\sum_{v \in V} h_v^{(L)} \tag{2}$$

where h_G represents the final graph-level embedding, $|V|$ denotes the total number of nodes in the graph, v represents individual nodes, and $h_v^{(L)}$ is the final node embedding for node v after L message passing layers.

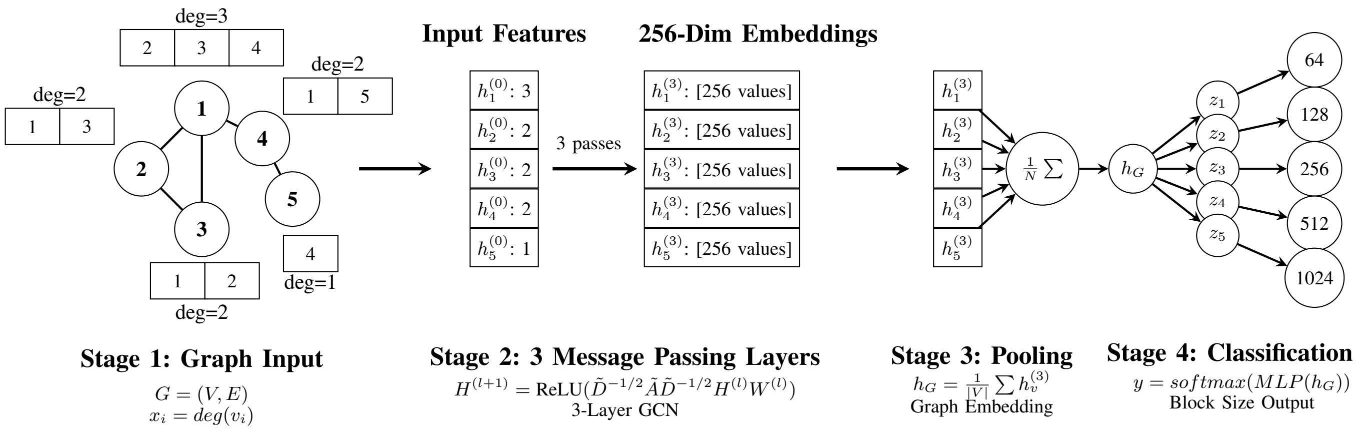

Fig. 4: **GCN Architecture for CUDA Block Size Prediction.** The model processes graph structure through four stages: (1) Graph input with degree features, (2) Three-layer GCN for message passing, (3) Global pooling for graph representation, and (4) MLP classifier for predicting optimal block size (64, 128, 256, 512, or 1024 threads).

The final component is a Multi-Layer Perceptron (MLP) that maps the graph embeddings to block size predictions. This MLP consists of two fully connected layers with ReLU activation, resulting in a 5-dimensional output representing the probability distribution over 5 possible block sizes (64, 128, 256, 512, and 1024 threads). CUDA supports block sizes in increments of 32 threads, up to a maximum of 1024 threads, as shown in [16] and [17]. However, for computational and experimental simplicity we have used 5 different block sizes (powers of 2) to separate our classes.

We formulated block size prediction as a classification task with one-hot encoded targets representing optimal block sizes. For each graph in our training set, we pre-computed execution times using block sizes of 64, 128, 256, 512, and 1024 threads. The block size yielding the fastest execution was encoded as the target class. This classification approach allows our model to directly predict the optimal categorical block size rather than attempting to estimate exact execution times.

For training, we employed the cross-entropy loss function shown in equation 3:

$$\mathcal{L} = -\sum_{i=1}^{5} y_i \log(\hat{y}_i) \tag{3}$$

where $\mathcal{L}$ represents the cross-entropy loss, i indexes the five possible block size classes (64, 128, 256, 512, 1024), y_i is the ground truth one-hot encoded label (1 for the optimal block size, 0 otherwise), and $\hat{y}_i$ is the predicted probability for block size class i output by the MLP classifier.

E. Scalable Training Strategies

A critical component of our approach is the neighbor sampling technique during message passing, based on the methodology introduced by [18]. Instead of processing the entire graph at once, which becomes computationally infeasible for large-scale graphs, this technique employs a mini-batch training strategy with neighbor sampling that addresses the fundamental challenges of training on graphs with vastly different scales and characteristics.

Scalable Neighborhood Sampling. For each node in a randomly sampled mini-batch, we sample a fixed number of neighbors at each message passing [18]. Including all neighbors overwhelms the message passing process for massive graphs and creates computational bottlenecks. The sampling strategy maintains consistent computational complexity across graphs of varying densities while preserving the essential structural information needed for accurate block size prediction. We process 32 neighbors in the first and second hop, and use 30 mini-batches per graph during each epoch, as this quantity provides statistically sufficient coverage of the graph structure.

Adaptive Batch Size Selection. Our mini-batch approach adjusts batch size based on graph characteristics to ensure balanced information extraction across our diverse dataset. Through empirical experimentation, we determined to assign graphs with over 5 million nodes batch sizes of 8, while smaller networks can accommodate batch sizes up to 128, enabling more aggressive sampling from their complete structure. These batch size thresholds were established through iterative testing to optimize the trade-off between computational efficiency and training stability across our diverse graph collection.

Gradient Normalization to Prevent Model Overfitting. To ensure balanced learning across graphs of different scales, we implement a gradient normalization technique that addresses the inherent imbalance in training contributions from graphs of varying sizes. Since larger graphs contain significantly more nodes and edges, each mini-batch from a massive graph processes substantially more structural information and requires more computations compared to mini-batches from smaller graphs. This computational disparity means that large-scale graphs naturally contribute stronger gradient updates and dominate the learning process, potentially causing the model to overfit to the characteristics of massive networks while underrepresenting smaller graph patterns. Our normalization approach scales the loss by the ratio of a graph size.

Alternative Training Configurations and Insights. Implementing the neighbor sampling strategy combined with gradient normalization and adaptive batch sizing techniques yielded the best performance results amongst our experiments. Initial attempts with larger batch sizes and unlimited neighbor sampling led to training times extending to weeks with poor generalization performance. Conversely, overly conservative approaches with minimal sampling resulted in underfitting, where models lacked sufficient structural information to make accurate predictions. The specific hyperparameters presented (neighbor sampling counts of 32-32, batch sizes of 8-128, and 30 mini-batches per graph) represent our optimized configuration derived through systematic experimentation across multiple parameter combinations. While these values provide strong performance across our diverse graph collection, they constitute one effective point in the hyperparameter space rather than theoretical optima. Alternative configurations may yield improved results for specific graph domains or hardware architectures.

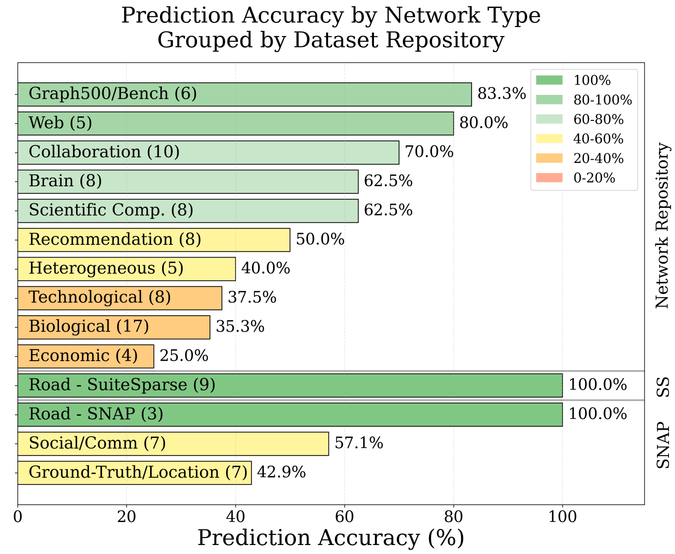

Fig. 5: Prediction accuracy by network type across three major graph datasets, Network Repository, SuiteSparse (SS), and SNAP. Numbers in parentheses indicate number of networks for each category. Performance varied significantly across categories while road networks achieved perfect accuracy due to their regular topology.

V. Experimental Evaluation

A. Experimental Setup

The triangle counting algorithm was implemented using C++ and the KOKKOS framework for high-performance computing. [19] The GNN model was implemented using PyTorch, PyTorch Geometric (PyG) and NetworkX frameworks. [15] The model was trained on NVIDIA A30 GPUs.

We divided our dataset into training and validation sets, with approximately 80% used for training and 20% for validation. We selected validation graphs that represent diverse topologies and sizes.

B. Performance Analysis

Figure 6 presents confusion matrices analyzing our model's CUDA block size prediction accuracy across overall, validation, and training datasets. The matrices show that our model achieves overall accuracy of 59.05% (62/105 correct predictions), with validation performance demonstrating 63.64% compared to 57.83% on training data. These results significantly outperform random selection (20% expected accuracy with 5 block size options). The diagonal elements highlight successful predictions, with the model showing particularly strong performance in predicting 64-thread blocks (35/37 correct overall) and effective accuracy for 512-thread blocks (14/27 correct overall).

The model demonstrates great prediction patterns with most errors occurring between adjacent block sizes as shown in the off-diagonal elements, indicating that incorrect predictions are typically close to optimal values. Predicting the adjacent block size yields a comparable performance as in the most cases performance of adjacent block sizes are close to the performance of the most optimal block size. The distribution pattern suggests the model has learned meaningful relationships between graph characteristics and optimal CUDA execution parameters

The unequal class distribution observed in our dataset (with 64-thread blocks being most common) reflects the natural tendency of many graph datasets to favor smaller block sizes for optimal performance. While we aimed to maximize network diversity across different graph types and topologies, this resulted in an imbalanced distribution where 64-thread configurations were optimal for a larger proportion of our collected networks. Similarly, we have fewer data samples from the 1024-thread class, as a smaller number of graph datasets naturally exhibit 1024 as their optimal block size

Overall
105 samples

A/P	64	128	256	512	1024
64	**35**	2	0	0	0
128	9	**8**	0	1	0
256	1	9	**6**	0	0
512	2	7	4	**14**	0
1024	0	0	3	4	**0**

Validation
22 samples

A/P	64	128	256	512	1024
64	**8**	1	0	0	0
128	2	**2**	0	0	0
256	0	3	**2**	0	0
512	0	1	0	**2**	0
1024	0	0	0	1	**0**

Training
83 samples

A/P	64	128	256	512	1024
64	**27**	1	0	0	0
128	7	**6**	0	1	0
256	1	6	**4**	0	0
512	2	6	4	**12**	0
1024	0	0	3	3	**0**

Fig. 6: Confusion matrices showing overall, validation, and training set performance for CUDA block size prediction. Overall accuracy: 59.05% (62/105), Validation accuracy: 63.64% (14/22), Training accuracy: 57.83% (48/83). Green cells indicate correct predictions, red cells show incorrect predictions, and gray cells represent zero predictions.

configuration.

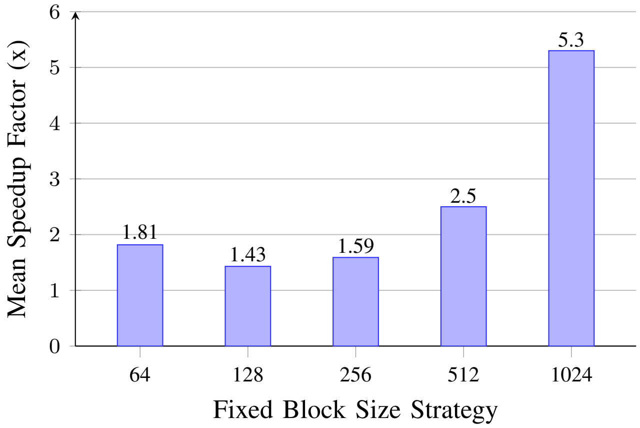

Fig. 7: Speedup factors achieved by our GCN model predictions compared to fixed block size strategies across all datasets.

Notably, our model struggles to predict 1024-thread blocks, achieving zero correct predictions in this category. This difficulty likely stems from the similar degree distributions between graphs optimal for 512, 256, and 1024-thread configurations, forcing our model to favor some classes over others. The overlapping topological characteristics of these larger block size categories suggest that more sophisticated graph features may be necessary to capture the subtle structural differences that influence optimal thread configuration.

The predictive power of our GNN model confirms that graph topology has a consistent and learnable relationship with optimal CUDA execution parameters. This proves the wisdom that optimal block sizes are determined by a combination of graph topology, hardware specifications and kernel characteristics. The effectiveness of our simple degree-based feature representation highlights the critical role of degree distribution in determining execution characteristics. Graphs with similar degree distributions often benefit from similar block size configurations, suggesting that degree-based clustering could serve as an effective heuristic for block size selection.

VI. Conclusion and Future Work

In this paper, we presented a GCN-based approach for predicting optimal CUDA block sizes for graph algorithms. Our model achieves 63% prediction accuracy across diverse graph topologies, translating to significant performance improvements compared to existing approaches. Experimental results demonstrate substantial speedup gains, with our model achieving up to 24.25× performance improvement over fixed block size strategies, and mean speedups ranging from 1.43× to 5.3× across different fixed configurations. By leveraging the structural properties of graphs through GCNs, our approach enables automatic tuning without the overhead of exhaustive empirical search. Future research directions could explore methods to improve the prediction accuracy while considering the level of complexity of the model, expanding the training dataset to include more diverse graph topologies and investigating other possible training strategies that improve the prediction precision and generalization capabilities. By bridging the gap between graph learning and HPC, our work demonstrates the potential of machine learning approaches to solve complex system optimization problems, particularly for irregular and data-dependent computation patterns.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Number 2323533. The work is also partially supported by the NSF MRI Award # 2117941 for GPU Cluster support.

References

[1] S. Arifuzzaman, H. S. Arikan, M. Faysal, M. Bremer, J. Shalf, and D. Popovici, “Unlocking the potential: Performance portability of graph algorithms on kokkos framework,” in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2024, pp. 526–529.

[2] “Cuda c++ programming guide,” https://docs.nvidia.com/cuda/cuda-c-programming-guide/#maximize-utilization.

[3] M. S. Rehman, “Exploring irregular memory access applications on the gpu,” in *Proceedings of the 2010 International Conference on Computer Design*. IEEE, 2010, pp. 513–520.

[4] J. Li, Z. Xu, M. Pham, Y. Tu, and Q. Zhou, “A comparative study of intersection-based triangle counting algorithms on gpus,” in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2024, pp. 1070–1081.

[5] S. Arifuzzaman, M. Khan, and M. Marathe, “Fast parallel algorithms for counting and listing triangles in big graphs,” *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 14, no. 1, pp. 1–34, 2019.

[6] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the gpu using cuda,” in *High Performance Computing – HiPC 2007*, S. Aluru, M. Parashar, R. Badrinath, and V. K. Prasanna, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 197–208.

[7] S. Sondur, A. Alazzawe, and K. Kant, “Optimal configuration of high performance systems,” in *The 2020 International Conference on High Performance Computing Simulation (HPCS)*, 2020.

[8] S. Zhang, H. Tong, J. Xu, and R. Maciejewski, “Graph convolutional networks: a comprehensive review,” *Computational Social Networks*, vol. 6, no. 1, pp. 1–23, 2019.

[9] D. Mukunoki, T. Imamura, and D. Takahashi, “Automatic thread-block size adjustment for memory-bound blas kernels on gpus,” in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. IEEE, 2016, pp. 377–384.

[10] T. A. Connors and A. Qasem, “Automatically selecting profitable thread block sizes for accelerated kernels,” in *2017 IEEE 19th International Conference on High Performance Computing and Communications*. IEEE, 2017, pp. 442–449.

[11] X. Cui and W.-c. Feng, “Iterml: Iterative machine learning for intelligent parameter pruning and tuning in graphics processing units,” *Journal of Signal Processing Systems*, vol. 93, pp. 391–403, 2021.

[12] M. Niepert, M. Ahmed, and K. Kutzkov, “Learning convolutional neural networks for graphs,” in *International conference on machine learning*. PMLR, 2016, pp. 2014–2023.

[13] H. Gao, Z. Wang, and S. Ji, “Large-scale learnable graph convolutional networks,” in *24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 1416–1424.

[14] X. Liu, J. Chen, and Q. Wen, “A survey on graph classification and link prediction based on gnn,” *arXiv preprint arXiv:2307.00865*, 2023.

[15] A. Hagberg, P. J. Swart, and D. A. Schult, “Exploring network structure, dynamics, and function using networkx,” Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[16] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, “ubench: exposing the impact of cuda block geometry in terms of performance,” *The Journal of Supercomputing*, vol. 65, pp. 1150–1163, 2013.

[17] I. Guerfi, L. Kriaa, and L. A. Saïdane, “Towards automatic block size tuning for image processing algorithms on cuda.” in *ICSOFT*, 2022, pp. 591–601.

[18] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” *Advances in neural information processing systems*, vol. 30, 2017.

[19] “GitHub - kokkos/kokkos: Kokkos C++ Performance Portability Programming Ecosystem: The Programming Model - Parallel Execution and Memory Abstraction — github.com,” https://github.com/kokkos/kokkos, [Accessed 29-05-2025].