Modeling Input Sensitivity in GPU Graph Analytics with Machine Learning

Jacob Hope[1]
Trisha Nag[2]
Dr. Apan Qasem[1]

[1]Texas State University | [2]Texas Tech University | National Science Foundation

We thank the National Science Foundation for funding the research under the Research Experiences for Undergraduates site programs (CCF-1659807) at Texas State University to perform this piece of work and the infrastructure provided by a NSF-CRI 1305302 award.

Problem Statement

Can machine learning derive optimization configurations that yield consistent performance on graph algorithms across different inputs?

Motivation

Graph algorithms are at the core of data-intensive applications in many computational domains including cybersecurity, medical informatics, business analytics and social data mining. This poster presents a machine learning based approach to capturing the irregular nature of GPU graph analytics. We conduct an extensive study of 1238 real-world graphs from different domains, each characterized by 70 distinct attributes [1].

Unlike previous efforts at ML-based performance modeling [2, 3], we take a ground up approach, and identify specific sources of input-sensitive performance inefficiencies in GPU graph algorithms, including two that have not been studied in previous research: (i) register pressure and (ii) CPU-GPU data movement via demand paging. We establish methods to address these inefficiencies at the compiler, system and algorithmic layers. The solution methods are parameterized to expose control knobs to a machine learning framework. We then build a classifier to characterize the relationship between graph attributes and a collection of control parameters, called a configuration. We build a system around the result in a classifier, which when given a new input graph and a GPU graph application, generates a kernel with an optimal configuration to mitigate the performance inefficiencies.

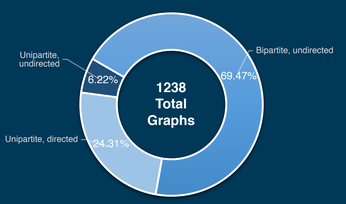

Figure 1. Dataset Graph Types

Graph Cardinality	1238
Graph Attributes	70
Performance Inefficiencies	Register Pressure, Thread-Block Size, Data Movement, Memory Divergence
Hardware Performance Counters Events	495 Total, 327 Metrics, 168 Events
GPU	NVIDIA Pascal Titan X, CUDA 10.0
Range of Register Counts	16, 24, 32, 64
Range of Kernel Launch Bound	32, 64, 128, 256, 512, 768, 1024

Table 1. Table of Configuration

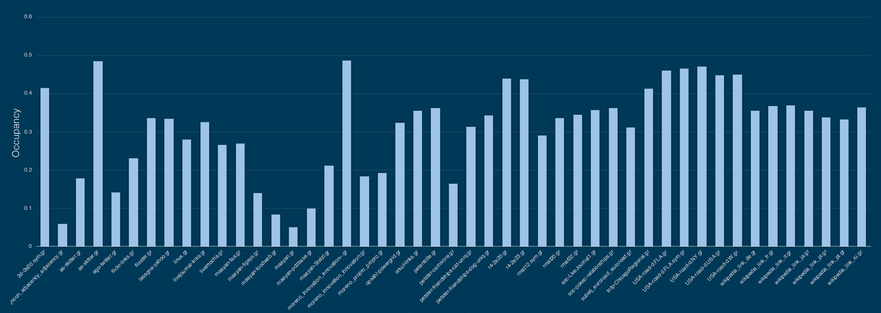

Figure 2. Input Sensitivity

References

[1] The KONECT Project. Retrieved 07/01/2019 from http://konect.cc/

[2] Sanath Jayasena, Saman Amarasinghe, Asanka Abeyweera, Gayashan Amarasinghe, Himeshi De Silva, Sunimal Rathnayake, Xiaoqiao Meng, and Yanbin Liu. 2013. ***Detection of false sharing using machine learning***. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13). ACM, New York, NY, USA, , Article 30 , 9 pages.

[3] Connor Imes, Steven Hofmeyr, and Henry Hoffmann. 2018. ***Energy-efficient Application Resource Scheduling using Machine Learning Classifiers.*** In Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018). ACM, New York, NY, USA, Article 45, 11 pages.

Methodology

(1) Identifying input-sensitive performance inefficiencies: Many different forms of performance inefficiencies manifest in GPU graph analytics. Not all are sensitive to input graph properties, however. In this work, we identify those performance inefficiencies that are impacted by variations in the input graph.We begin with a set of GPU performance bottlenecks that have been previously studied in the literature [2, 3]. We consider two additional ones that have not been explored previously in the context of GPU graph analytics: (i) register pressure and (ii) CPU-GPU data movement via demand paging. We characterize and quantify each performance inefficiency in terms of a set of GPU hardware performance counters.

(2) Selecting graph attributes to match performance inefficiencies: We characterize each graph in our database in terms of 70 distinct attributes. From these we select a subset of attributes that have the most influence on the performance inefficiencies identified in Step 1. To achieve this, we build another set of regression

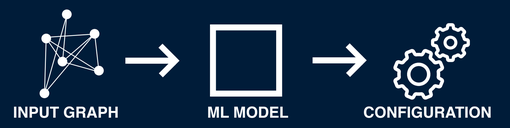

Figure 3. High-Level Abstraction of Methodology

models, one for each selected performance inefficiency, with graph attributes as the independent variables and the hardware performance counter values as the dependent variable. From each model, we select the top four candidates and then combine them to derive a set of k attributes.

(3) Exposing tunable control knobs: For each inefficiency identified in Step 1, we find methods to address them. These methods include compiler techniques (register allocation), runtime systems (host and device memory allocation) and algorithmic methods (topology vs datadriven algorithms). For each method, we identify a set of tunable parameters that are the main controlling force behind the solution. For example, we parameterize the CUDA register allocator by the maximum number of registers to be used in each thread.

(4) Mapping graph attributes to tunable control knobs: We then construct a decision tree classifier to map an input graph, represented by the set of attributes selected in Step 2, to the optimal configuration of control knobs. In the training set, the optimal configuration for each graph is discovered via autotuning. For a given input graph the autotuner explore the space of all feasible configurations and discovers the one that yields the highest overall performance.

(5) Kernel generation: Finally, we integrate the classifier into a system which, given a new unseen graph, extracts the relevant features, predicts the optimal configuration and generates a new kernel with the optimal configuration.

Results

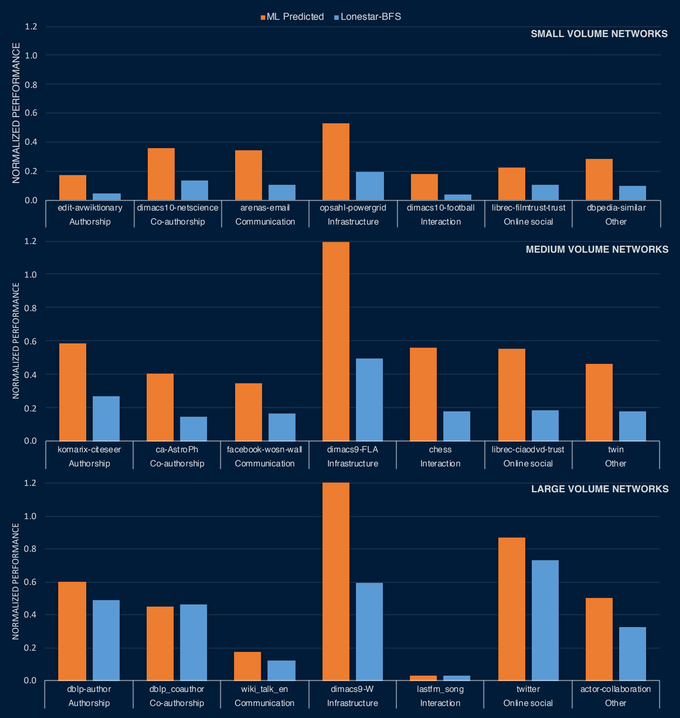

Figure 4. IPC Comparision of ML Model Predicted vs Lonestar-BFS Default Configuration

ML MODEL'S BEST PREDICTION RESULTED IN A ***4.48X SPEEDUP***

ML MODEL'S AVERAGE PREDICTION RESULTED IN A ***1.82X SPEEDUP***

Accuracy	0.891
Precision	0.887
Recall	0.891
F1 Score	0.888

Table 2. Stratified 10-Fold Cross-Validation Averaged Results

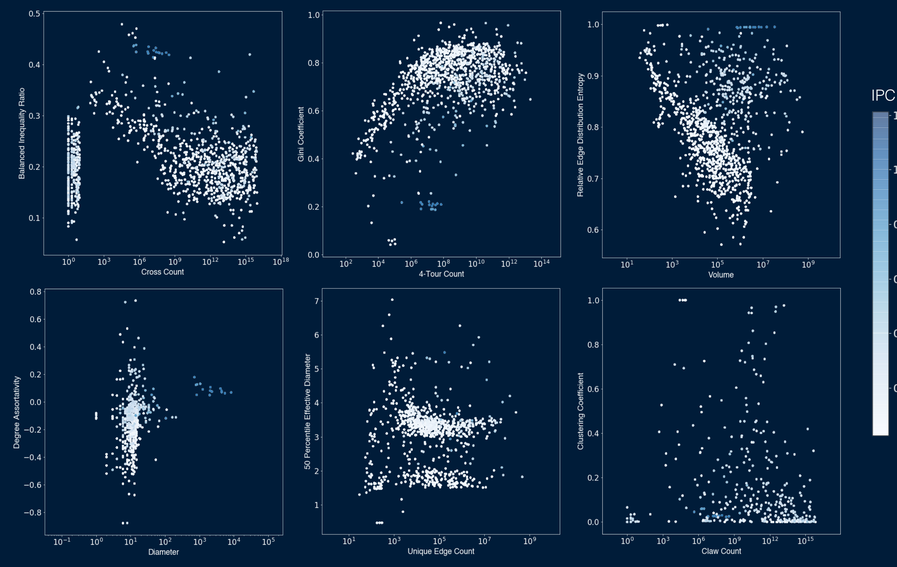

Figure 5. Non-Linearity of Graph Features and Their Relationship with IPC

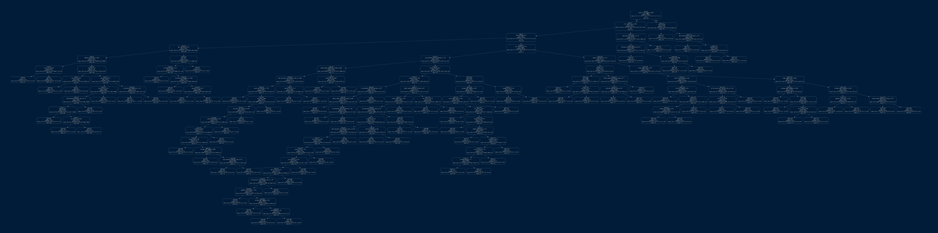

Figure 6. Decision Tree Classifier from 3rd Fold

Conclusions

An ML-based strategy can provide substantial performance improvements over highly-tuned state-of-the-art BFS implementations, achieving an average speedup of 1.82 and up to a 4.48 speedup on certain classes of input graphs.

The experimental results show that performance bottlenecks stemming from inefficient utilization of the GPU register file and the demand paging mechanism are sensitive to the structure of the input graph. An input-oblivious demand paging scheme can result in a factor of 47.5 performance variation across graphs. The input-sensitivity of the register allocation scheme is less severe but still significant with a factor of 40 performance variation across all inputs.

Post-analysis of our ML model reveals that (i) degree of distribution (ii) edge density and (iii) graph diameter can all have significant on performance and GPU kernel optimization choices.