Characterizing Input-sensitivity in Tightly-Coupled Collaborative Graph Algorithms

Jacob M. Hope
Skygrid
Austin, TX
jacobmhope@gmail.com

Mikel Gjergji
University of Rhode Island
Kingston, RI
mikel_gjergji@uri.edu

Johana Di Girolamo
Texas State Univ.
San Marcos, TX
j_d537@txstate.edu

Marco Alvarez
University of Rhode Island
Kingston, RI
malvarez@cs.uri.edu

Apan Qasem
Texas State Univ.
San Marcos, TX
apan@txstate.edu

***Abstract*—This paper conducts a study of input-sensitivity in collaborative graph algorithms for CPU-GPU systems with support for Unified Memory. The study, conducted on an extensive set of real-world graphs from the Koblenz Network, identifies three main sources of performance inefficiencies that are influenced by characteristics of the input graph. We develop autotuning methods to specifically address these inefficiencies. We then explore machine learning approaches to characterize the relationship between input graph properties, performance, and optimization parameters. In applying our learned models to a test dataset of 70 real-world graphs on the problems of breadth-first search (BFS) and single-source shortest path (SSSP), we are able to attain 96.33% of the peak performance on BFS, and 99.40% on SSSP when using the top-3 predictions of a neural network. We also attain 95.63% of the peak performance on BFS when using three decision trees trained on different categories of graphs. The performance of the learned models is superior to selecting the most frequent optimal configuration, indicating that the machine learning models were able to successfully correlate our selected configuration parameters with graph attributes.**

***Index Terms*—heterogeneous memory; graph processing; machine learning; code optimization**

I. Introduction

Graph algorithms are at the core of data-intensive applications in many computational domains, including cybersecurity, medical informatics, business analytics and social data mining. As such, efficient graph processing is of critical importance. Recent research has shown that highly-tuned, massively parallel GPU graph analytics can yield impressive results [1]–[4]. Yet, the irregular structure of graphs continues to be a major obstacle in unleashing the full capabilities of the underlying hardware. Irregularity in real-world graphs can make performance unpredictable and non-portable across different inputs and architectures [5]–[7]. Depending on the type of graph being processed, the same optimized implementation of an algorithm can produce performance numbers that differ by *orders* of magnitude.

Emerging trends in application development further exacerbate the challenges with irregularity and input dependence. Industry vendors have recently introduced technology that presents a unified view of multiple physical pools of memory contained within a compute node [8]–[10]. In Unified Memory(UM) systems, pointers can be freely used between different memory regions in the CPU and GPU, relieving developers from the burden of explicitly managing data. Improved programmability have made UM an attractive choice for high-end systems, including those that are set up to use NUMA [11], [12]. Application developers are taking advantage of UM to create new collaborative design patterns that emphasize tight coupling of CPU-GPU tasks [13], [14]. These emerging paradigms can mitigate GPU oversubscription, a ubiquitous problem in large-scale graph processing [15], [16]. Although the programmability benefits are substantial, the collaborative paradigms make reasoning about performance issues in irregular codes more complicated. The degree of collaboration must be carefully orchestrated considering the sparsity of access and the size of the partitions, making performance more intimately tied with the structure of the input graph [12], [17]. Furthermore, in these collaborative paradigms, segments of the graph may be concurrently shared between the CPU and GPU. This not only requires tighter synchronization windows but also makes it necessary to consider the differences in the memory hierarchies in the CPU and GPU and how they cater to the structure of the input graph.

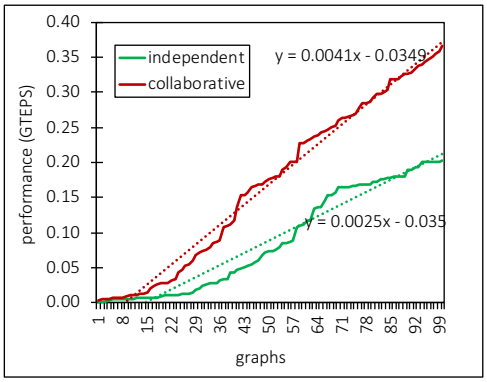

Fig. 1: Input sensitivity comparison of two BFS implementations [Intel Xeon, NVIDIA Pascal, CUDA 10.0]

Fig. 1 shows performance variations for two BFS implementations across 100 directed acyclic graphs. The collaborative BFS implementation utilizes demand paging and employs a CPU-and-GPU-Iterative design pattern [14]. Although this implementation achieves significantly higher performance on almost all input graphs, it also exhibits higher degrees of input sensitivity than the non-collaborative version. The best performance point is $91\times$ better than the worst one. These numbers reiterate the need for investigating input-sensitivity

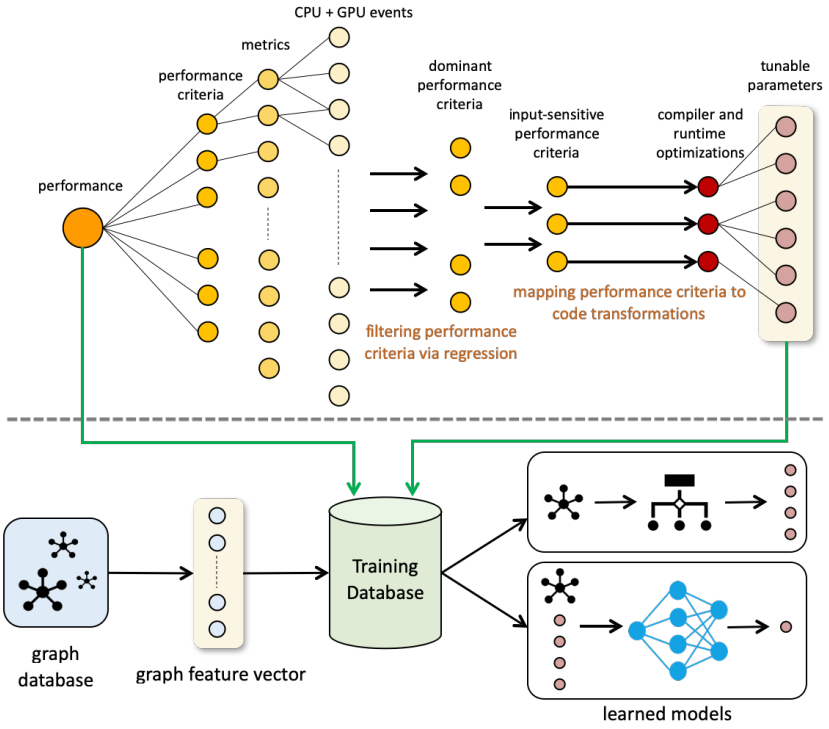

Fig. 2: A bottom-up data-driven approach to exploring input sensitivity graph algorithms

in emerging paradigms.

This paper explores input-sensitive behavior of heterogeneous graph algorithms running on CPU-GPU systems that support Unified Memory. The study is conducted on a corpus of 1000 real-worlds graphs collected from 21 computation domains [18]. The structure of the graphs are captured using a set of 67 distinct attributes. The study primarily focuses on two widely-used and important graph algorithms: Breadth-First Search (BFS) and Single-Source Shortest Path (SSSP). We select BFS and SSSP because (i) they are representative of the computation domains studied (ii) both serve as building blocks in many important classes of applications such as cycle detection, maximum flow, and betweenness centrality and (iii) implementations of both algorithms are featured in the latest high-performance graph frameworks such as GraphIt [4], Gunrock [3] and Gaolois [19].

In the first phase, we conduct a performance study to identify performance characteristics of collaborative graph applications that are sensitive to the properties of the input graph. Unlike prior studies however, we take a ground-up, data-driven approach in which we attempt to automatically discover the pertinent performance bottlenecks via experimentation and statistical analysis. This approach, described in Section II, generalizes to other contexts in a straightforward manner. The experimental study identifies (i) access density (ii) thread-level parallelism, and (iii) register space utilization as the three most important criteria that are sensitive to input graph properties.

In the second phase, we develop parametric methods to address the inefficiencies that can result from the three performance criteria. The methods include a combination of source-to-source, compiler and runtime optimizations. For each method, we expose tunable control parameters and build an autotuning system. The autotuner is used to generate a training database of cross-input performance data. We then evaluate two machine learning approaches, using decision trees and neural networks, to characterize the relationship between input graph properties, the optimization parameters, and performance. We build a system around the resulting classifiers, which when given a new input graph and an implementation, generates a kernel with the optimal configuration. This work yields the following results

- in addition to size and volume, (i) degree distribution (ii) edge density, and (iii) graph diameter can all have significant impact on performance and optimization choices;
- performance inefficiencies stemming from (i) increased data movement under UM, (ii) unexploited parallelism, and (iii) inefficient utilization of the GPU register space are influenced by the structure of the input graph;
- the relationship between graph structure and performance can be learned. This learning can be embedded in code optimizations which can yield integer factor performance improvements over state-of-the-art implementations.

II. Approach

Fig. 2 provides an overview of our approach to exploring input sensitivity in graph algorithms. Our approach is data-driven and relies heavily on extensive profiling via hardware performance counters. Below, we outline the major steps.

1. Identifying input-sensitive performance inefficiencies: Performance of irregular graph algorithms can be limited by

TABLE I: Performance domains, criteria, metrics and events for characterizing performance of irregular CPU-GPU codes.

Domain	Criteria	Metric	Events
Data Movement	Data volume	copy-to-comp ratio	CUDA_H2D_memcpy, CUDA_D2H_memcpy, ...
	Access pattern	access density	ldst_executed, inst_compute_ld_st, ldst_issued
	Oversubscription	page re-migration rate	total_bytes_transferred_over_PCIE_H2D, total_bytes_transferred_over_PCIE_H2D, total_bytes_read_from_DRAM_to_L2, total_bytes_written_from_L2_DRAM
Achieved Parallelism	Thread-level parallelism	occupancy	average_active_warps_per_cycles, max_warps_supported_per_SM,...
	Instruction-level parallelism	attained ILP	issue_slot_utilization, warp_execution_efficiency,...
	Control divergence	branch efficiency	non-divergent_branches, total_branches
		predication rate	thread_inst_executed, not_predicated_off_thread_inst_executed
	Pipeline utilization	execution stall rate	stall_exec_dependency, inst_executed, inst_issued
		memory stall rate	stall_memory_dependency, inst_executed, inst_issued
		sync stall rate	stall_sync, inst_executed, inst_issued
		fetch stall rate	stall_inst_fetch, inst_executed, inst_issued
Memory Hierarchy Utilization	Memory coalescing	load divergence	global_load_transactions, global_load_requests
		store divergence	global_store_transactions, global_store_requests
	Cache behavior	L1 miss rate	unified_l1_hit_rate, inst_executed
		L2 read miss rate	l2_requests_from_tex_cache,l2_hits_for_tex_cache_requests, inst_executed
		L2 write miss rate	l2_tex_cache_write_hit_rate, inst_executed
	Register space utilization	register efficiency	tex0_cache_sector_queries, tex1_cache_sector_queries, register spills, registers per thread, ...

many factors. Not all of these are sensitive to input graph properties. To identify input-sensitive performance bottlenecks, we first identify a broad range of performance characteristics observed in CPU-GPU hybrid applications. We call this set $B = \{b_1, ...b_N\}$. For each performance characteristic b_i, we establish a set of metrics $\{m_1, ..., m_j\}$ to quantify program behavior in each dimension. Each metric in turn is expressed as a combination of CPU and GPU performance events.

$$B = \{b_1, ..., b_N\} \approx \{\{m_1, ..., m_j\}, ..., \{m_1, ..., m_k\}\}$$

We then construct a regression model $\mathcal{R}$, to correlate specific inefficiencies with overall performance P.

$$\mathcal{R}(< \{m_1, ..., m_j\}, ..., \{m_1, ..., m_k\}, P >) \implies B'$$

From $\mathcal{R}$, we derive a set $B' \subseteq B$, which includes the inefficiencies that are most tightly related to overall performance. We further analyze the data to obtain a set $B'' \subseteq B'$ which represents the critical performance inefficiencies that are most input sensitive.

2. Exposing tunable control parameters: For each $b \in B''$, we identify a method to address the inefficiency. These methods include source-to-source, compiler and runtime code optimizations. The code transformation techniques are then parameterized to expose a set of tunable parameters.

$$B'' = \{< t_1, ..., t_q >, ..., < t_1, ..., t_r >\}$$

We exhaustively evaluate two graph algorithms in this parameter space for all input graphs in the dataset. The performance is recorded and added to the training database.

3. Mapping graph attributes to optimal tuning parameters: The size and structure of a graph are defined by using a set of 67 distinct attributes. These attributes are extracted via a single pass over the edge-list representation of the graph. To model the relationships between graph attributes, optimizations, and performance, we evaluate two machine learning approaches. Both approaches are illustrated under the 'learned models' section in Fig. 2.

The first approach involves training a decision tree to learn a direct mapping from an input graph, represented by a flat vector of attributes, to the optimal configuration of optimizations. The decision tree is constructed primarily to derive insight about how input graph properties impact the performance characteristics. A second approach involves training a neural network to learn the mapping from the pair [graph attributes, configuration parameters] to a performance metric, e.g. TEPS (traversed edges per second). The neural network enables the prediction of performance for multiple configurations.

III. Characterizing Irregular Performance

We perform a hierarchical analysis [20] to characterize and quantify input sensitivity in irregular codes. First, we break down overall performance into three *domains*: data movement, achieved parallelism and memory hierarchy utilization. The three areas represent aspects of program behavior that can be the source of potential performance inefficiencies in irregular CPU-GPU applications [14]. We then categorize performance in each dimension into a set of performance *criteria*. We derive metrics to quantify performance in each criteria and finally, we formulate methods to measure each metric using a combination of available CPU and GPU performance events. Table I lists the performance domains, criteria, metrics and events. Next, we discuss the analysis in the context of collaborative graph algorithms and describe the methodology for measuring the metrics. In the interest of space, we limit the discussion to only the key elements in Table I.

A. Data Movement

The volume of data transferred between the CPU and the GPU has a major impact on the performance of hybrid applications. Data movement becomes even more critical under UM systems. In a UM system, data can reside either in host or device memory during GPU kernel execution and the placement choice that minimizes data transfer times depends on the computation and data access patterns of the particular

application. Accessing host-resident data via demand paging is expensive. Data is fetched over a high-latency, low-bandwidth channel and page migrations incur additional overhead due to fault handling. Notwithstanding, the massively multi-threaded GPU kernels can potentially hide some fraction of the latencies and mitigate the costs associated with demand paging, making placement decisions non-obvious. For irregular applications, data placement choices become more complicated for the following reasons:

a) Sparse data access: With demand paging, only the data requested by the GPU is transferred from host memory. As a result, for graph applications that exhibit sparse or irregular data access patterns, the actual volume of traffic over the interconnect can be a lot smaller under demand paging than bulk-copy. To minimize data traffic, we need to estimate *access density* which is dependent on the input graph properties.

b) Oversubscription: Real-world graph analytics typically deal with very large graphs that oversubscribe GPU device memory. Under UM, data can be kept in host memory to mitigate oversubscription. Demand paging eliminates the need for strip-mining and can provide performance by reducing the overhead of repeated kernel launches. Nonetheless, this decision must be made on a *per-input* basis, considering the size and dimensions of the graph.

To quantify performance inefficiencies with respect to data movement we introduce the following metrics.

Copy-to-computation ratio is computed as the ratio of data copy time to the kernel execution time. Intuitively, this metric represents the cost to the application due to bulk-copying of data from host memory.

Access density denotes the fraction of data that is explicitly requested by the compute elements in the GPU. If every element in the data structure is touched at least once (i.e., dense access) then density = 1. A lower value indicates higher access sparsity.

Page re-migration rate A page may need to be migrated more than once if it is evicted from memory before its reuse. In the traditional design pattern, re-migration only occurs from CPU to GPU. But in collaborative applications re-migration can be bi-directional. For a given data structure, the page re-migration rate is measured as the ratio of the total number of page migrations to the number of distinct pages requested by the compute elements. Each requested page is migrated at least once. Hence, re-migration rate is always ≥ 1.

B. Achieved Parallelism

GPU throughput is a direct result of the amount of parallelism exploited in the kernel. This includes both thread-level and instruction-level parallelism. The shape of the input graph can impact the attained level of parallelism. For instance, graphs with higher edge-density provide opportunities for greater parallelism in applications that use vertex partitioning. We quantify the amount of thread-level parallelism with *achieved occupancy* which is defined as the ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor. Instruction-level parallelism is measured as *attained ILP* which is a combination of two metrics, issue slot utilization and warp execution efficiency.

Complex control-flow can lead to branch divergence which can also inhibit parallelism. We quantify control divergence with two metrics: (i) *branch efficiency* which is the ratio of non-divergent branches to total branches expressed as a percentage and (ii) *predication rate* which is the ratio of the average active threads per warp executing non-predicated instructions to the maximum number of threads per warp.

C. Memory Hierarchy Utilization

1) Memory Coalescing: Uncoalesced memory access, a common occurrence in irregular applications, can cause severe performance degradation on the GPU. We quantify memory divergence as the number of global memory transactions per request, adjusted for the length of the coalescing unit. A value greater than one indicates application is suffering from some amount of divergence.

2) Cache Behavior: Although cache performance is not as critical for irregular applications on the GPU, it has been shown that locality does exist and can have a significant impact on performance [21]. Indirect memory addressing and memory access based on node values make it difficult to determine the amount of locality in irregular applications. Nonetheless, some forms of locality can be captured by properties of the input graph. For example, graphs with uniform edge distribution might exhibit better data locality.

3) Register Utilization: Utilization of the register file is critical for GPU performance. Effective allocation can eliminate spills and improve single-threaded performance. Since the register space is a shared resource, over allocation can limit the number of threads in a block and consequently reduce thread concurrency. In irregular applications, optimal register allocation is input dependent for the following reasons.

a) Launch Bounds: Kernel launch bounds can vary across input datasets. The allocation policy must take this into account. A smaller block size implies that the register allocator can be more aggressive while a larger block size merits a conservative approach.

b) Dependence chains: The diameter of an input graph determines the length of the dependency chains in the kernel. Kernels with longer chains, and consequently longer live ranges, will require a larger number of registers to avoid spills. This will increase the register pressure which in turn may limit the size of the thread blocks.

D. Pinpointing Input-Sensitive Performance Inefficiencies

Although each metric is important, in this work, we are primarily interested in identifying those criteria that are (i) most critical and (ii) most input-sensitive. To this end, we formulate a regression model in which each metric from Table I constitutes an explanatory variable while overall performance is designated as the response variable. Performance is measured as billions of traversed edges per second (GTEPS), which allows us to normalize across input graphs and across

different systems. We execute the graph algorithms on each input graph in the database and collect the metrics via hardware performance counter-based profiling. Because of multiplexing of performance counters, a total of 16 runs was necessary to collect all metrics. Each profile run was repeated 4 times and the average values were included in the regression dataset. Principal Component Analysis is applied to address issues with multicolinearity (e.g., L1 and L2 miss rates).

TABLE II: Regression statistics for identifying input-sensitive performance criteria. = metrics that are performance critical and input sensitive; = performance-critical but not input sensitive; = not performance-critical

	Metric	coeff	t	P[97.5]	variance
significant	occupancy	1.0471	8.486	0.000	1112.91
	access density	2.2555	3.930	0.000	2142.83
	load divergence	-0.9260	4.002	0.000	33.26
	L2 read miss rate	-0.9473	-12.782	0.000	118.36
	branch efficiency	0.0397	0.856	0.000	269.29
	register efficiency	0.9451	1.314	0.001	1716.05
	page re-migration	-0.0722	-2.007	0.022	17.46
insignificant	copy-to-comp	-0.5672	-11.413	0.034	
	predication rate	0.2294	5.103	0.045	
	attained ILP	0.5910	1.927	0.054	
	exec stall rate	0.1251	3.132	0.156	
	store divergence	-3.5798	-3.306	0.391	
	L2 write miss rate	-0.0024	-0.924	0.783	
	sync stall rate	0.0021	0.213	0.817	
	memory stall rate L1 miss rate fetch stall rate	eliminated via PCA			

The regression results are reported in Table II. The rows are sorted in ascending order of the P-value at 97.5% confidence interval. Of the 17 metrics, three were eliminated via PCA. Of the remaining 14 variables, seven are statistically significant. Of these, branch efficiency and page re-migration has little impact in determining overall performance. We eliminate these two from further consideration in our study.

We then analyze the data to determine the spread of values for each statistically significant metric. The spread, calculated as the variance, is the leading indicator of input sensitivity. We observe that although memory divergence is a significant determinant of performance, it is not impacted by variations in the structure of the input graph. We select the three metrics, highlighted in green, with the highest input sensitivity for the next phase of the study.

IV. Exposing Tunable Control Parameters

We identify three optimization techniques to mitigate the performance bottlenecks identified by the regression model: (i) data placement for access density (ii) launch bound selection for occupancy and (iii) register allocation for register efficiency. We parameterize the heuristics employed by each optimization and expose the tunable parameters for external control by an autotuner. The source-level transformations are implemented within LLVM [22]. The runtime system and the SASS register allocator are implemented as standalone tools.

```
cudaMalloc(dev_ptr, ...);
cudaMemCpy(dev_ptr, host_ptr, ...);
kernel<<<grid_size,block_size>>>(dev_ptr)
cudaMemCpy(...);
```

(a) before

```
for d in D
   mem[d] = get_placement_auto();
// allocation site
if (mem[d] == dev)
  cudaMalloc(dev_ptr, ...);
else
   cudaMallocManaged(host_ptr, ...);
// copy-in site
if (mem[d] == dev)
   cudaMemCpy(dev_ptr, host_ptr);

unsigned int BLK = get_thread_block_auto();
if (is_legal(grid_size, BLK)) {
  if (mem[d] == dev)
    kernel<<<grid_size,block_size>>>(dev_ptr)
  else
    kernel<<<grid_size,block_size>>>(host_ptr)
}
else
  exit_gracefully();
// copy-out site
if (mem[d] == dev)
   cudaMemCpy();
```

(b) after

Fig. 3: Source-to-source transformations for exposing data placement and thread block size parameters for autotuning

A. Data placement

In UM systems, the penalty associated with data movement is largely determined by allocation decisions. On CUDA devices, data can be mapped to a managed space via the `cudaMallocManaged()` API. Data in the managed space is allocated to host and device memory using a first-touch policy. Our parameterization of data placement decisions involves a source-to-source transformation and a runtime system.

1) Source-to-source: As outlined in Fig. 3, each application goes through two source-level transformations.

(i) Parameterize placement of each data structure: For each data structure accessed by a GPU kernel, we identify the allocation and deallocation sites. Each site is bracketed with a placement-dependent clause and appropriate calls to the host allocator and deallocator are inserted in the host path. We wrap each copy site with a condition such that copy calls are only invoked when the data structure is allocated to device memory. Alternate launch configurations are added to the kernel launch site. For a kernel with n arguments, $2^n - 1$ alternate configurations are added.

(ii) Insert calls to the runtime system: The parameterized source is instrumented with calls to the runtime system. A single call is inserted at the allocation site for each data structure to obtain its placement location.

2) Runtime: The runtime system obtains placement parameters from the autotuning search algorithm. Each data structure in the application has a boolean-valued parameter for each kernel invocation. A *true* value indicates host placement and demand paging while *false* denotes device placement.

B. Launch bound selection

A pass over the application source replaces each kernel invocation with a new invocation in which the thread block size is replaced by a variable whose value is supplied by the autotuner at runtime. The source-to-source pass also instruments the code with the following runtime calls (Fig. 3).

(i) `get_thread_block_size()`: obtain the value of the thread block size from the autotuner
(ii) `is_legal()`: check the legality of the block size prescribed by the autotuner
(iii) `exit_gracefully()`: if the block size is deemed illegal then the program exits gracefully and communicates to the autotuner to retry the instance.

The block size parameter can take any integer value between one and the maximum threads allowed per block.

C. Register allocation

Designing an autotuning scheme for GPU register allocation is challenging for two reasons. First, it is not obvious which parameters should be exposed for external control. Second, the closed-form nature of CUDA SASS makes it difficult to implement a new allocation scheme. We address these issues by introducing the notion of *eagerness* in GPU register allocation. Intuitively, for a given kernel with a fixed register pressure, an *eager* allocator will attempt to allocate more, and usually longer and complex, live ranges to registers. By contrast, a *lazy* allocator will take a conservative approach and forego assignment of a live range even if a sufficient number of registers are available for allocation. Depending on the need, the system can be set up to support different levels of eagerness in the register allocation. In this study, we parameterize the CUDA register allocator on an *eagerness scale* which consists of the following levels: (i) conservative (ii) moderate (iii) aggressive. The autotuner sends the eagerness value to the compiler in the following way.

(i) Compilation flag: The `nvcc` compiler supports the flag `--maxrregcount REG` which can be used to limit the maximum number of registers allocated to a kernel. We map the eagerness value to the REG parameter. moderate maps to the default allocation; while conservative and aggressive map to $[default - r]$ and $[default + r]$, respectively.

(ii) SASS pass: We extract the SASS representation from the `nvcc`-generated binary using `turingas` [23] and implement a register assignment pass on the SASS which attempts to assign live ranges to registers that were skipped by the `nvcc` allocator. If the parameter value is conservative or moderate then no new registers are allocated. If the value is aggressive then it will attempt to allocate live ranges of size $\leq k$.

V. Applying Machine Learning

After identifying performance criteria with high input sensitivity and implementing techniques to control tunable parameters, we train machine learning models under two different approaches. The primary goal in both cases is to build a model that, given an unseen graph, it predicts the optimal configuration, which can be later used to generate a new and more efficient kernel.

A. Graph Dataset

The master dataset consists of 1000 real-world graphs obtained from the Koblenz Network Collection [18]. The graphs represent 21 domains, including social networks, biological networks and road networks. Edges and nodes capture the nature of relationships across these domains, including collaboration, communication, affiliation and trust. This dataset consists of three main classes of graphs (i) bipartite, undirected (ii) unipartite, directed, and (iii) unipartite, undirected. We refer to these classes as *bipartite*, *directed* and *undirected*, respectively. Graphs in each class can be either weighted or unweighted.

In our experiments with machine learning, we use a subset of the master dataset (676 graphs). In particular, to get more reliable performance data, we eliminated graphs that were very small (< 1000 nodes), and graphs for which it was not possible to extract all relevant features. Graphs were processed to extract a vector of 67 attributes for each graph. These attributes include statistics that describe the graph structure, mostly related to degree distribution, edge density, and graph diameter [18].

Using the autotuning methods presented in Section IV, each graph is annotated with its corresponding optimal configuration. Every configuration is a triplet of three tunable optimization parameters: data placement, thread block size, and register allocation. For finding the optimal configuration the autotuner explores the space of the following values for each parameter respectively: [`device, host`], [`64, 128, 256, 512, 1024`], [`conservative, moderate, aggressive`].

The subset of 676 graphs is split into training (606 graphs) and testing datasets (70 graphs). The training data is used for training all models and selecting their hyperparamenters. The test set is only used for calculating and reporting final accuracies. It is worth to note, that there is no intersection between the graphs in the training and test datasets. The training dataset contains 268 directed, 69 undirected, and 269 bipartite graphs. The test dataset contains 31 directed, 8 undirected, and 31 bipartite graphs.

B. Approach 1: Decision Trees

We train decision tree classifiers to learn the mapping of graphs, represented by their attributes, directly to their optimal configuration. The classifiers implement an optimized version of the C4.5 decision tree algorithm. We train three separate decision trees specific to each graph type: *bipartite*, *directed*, and *undirected*. In order to obtain the optimal configuration on a new graph, its attributes are fed through the correct decision tree according to the graph type, and the tree outputs the values of the predicted optimal configuration.

C. Approach 2: Multi Layer Perceptron

On a different approach, instead of taking as input graph attributes and learning to predict the optimal configuration, a

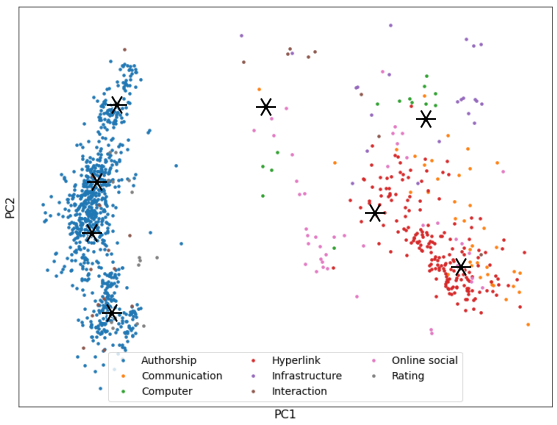

Fig. 4: Feature space visualization across different domains.

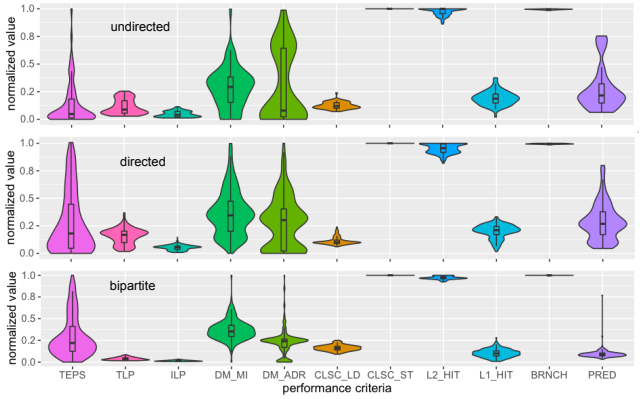

Fig. 5: Violin plots with kernel density and distribution for 10 performance criteria.

neural network is trained to predict the performance GTEPS. In this case, for each graph in our training and test datasets, 30 instances can be generated, increasing the size of the training dataset to 18180, and the size of the test dataset to 2100 instances. Each new instance is the concatenation of the graph attributes with a configuration triplet. Every instance is labeled with the corresponding GTEPS value found by the autotuning process. In this new formulation the neural network is trained with examples using all configuration types, instead of manually filtering the best configurations, as in the case of the decision tree classifiers.

VI. Experimental Results

A. Experimental Setup

1) Target Applications: The target applications in this study include topology-driven hybrid implementations of BFS and SSSP from the Galois graph optimization framework [19]. Both are optimized for CPU-GPU collaborative execution using Galois primitives. We consider these highly-tuned implementations as the *baseline*.

2) Evaluation Platform: All experiments were conducted on a compute node featuring a POWER8 system with 128 logical cores connected to two NUMA sockets. The node has four NVIDIA Tesla P100 GPUs connected to the CPU via NVLink. Experiments are conducted with CUDA 10.0 runtime and driver. All applications are compiled with `nvcc` at the highest optimization level. Each experimental run is repeated five times in both training and testing phases and only the average numbers are used in the study.

B. Graph Feature Space

To get better insight into the structure of graphs across different domains, we visualize the feature space using Principal Component Analysis (PCA) and k-means clustering. Fig. 4 plots the top two principal components (PC1 and PC2). Domains are color coded and clusters are shown using a $\star$. In addition to size and volume, the features represented in PC1 and PC2 include degree distribution (captured via the GINI coefficient), edge density and the graph diameter. We observe that PC1 features are sufficient to capture the structural differences among the three main categories of graphs, creating clear delineations *bipartite* (left), *directed* (right), and *undirected* (center). PC2 and the sub-clusters in each group further distinguish structurally similar graphs within each category. We also observe that there can be structural similarity between graphs that belong to different domains. For example, the clusters that form around the *Authorship* networks also include many graphs from the *Ratings* networks. Similarly, graphs that belong to the same domain may exhibit different structural properties. For example, graphs representing *Communication* networks appear in four different clusters. Finally, there might be graphs that belong to a particular domain that have their own unique features, as seen from the cluster around *Infrastructure* networks (top right corner).

C. Performance Criteria and Input Sensitivity

Fig. 5 visualizes the performance space of BFS across ten different criteria from Section III. As expected we see wide variation in overall performance (TEPS) across input graphs in all three categories. Interestingly however, many of the points are clustered in the bottom half, indicating that the BFS implementation yields poor performance for a large fraction of the input graphs. The most dominant performance criteria are the ones associated with data movement which shows considerable variation and closely resembles the TEPS plots. TLP, measured via occupancy, is significant for *undirected* and *directed* graphs but tends to have less of an impact on *bipartite* graphs. Branch efficiency and store coalescing proved to be input insensitive, both exhibiting favorable numbers for all input graphs. This is not surprising given the computation and data access patterns in BFS.

D. Model Prediction

In our experiments, decision trees and MLPs are trained using 10-fold cross validation for hyperparameter selection. We used the corresponding training set for each of the two graph algorithms, BFS and SSSP. The hyperparameters are fixed across all folds, and those that perform best are selected for the final model. The decision trees are fine-tuned by

trying different values for maximum depth, maximum features, minimum samples split, and split criteria. As separate decision trees are trained for different graph types, each has its own selected hyperparameters. The MLP hyperparameters were selected by a random search amongst different batch sizes, hidden layer sizes, learning rates, and L2 regularization values. The MLP for SSSP uses adam optimizer with alpha of 0.2355, learning rate of 0.0007, batch size of 128, and four hidden layers (80, 60, 40, 20). The MLP for BFS uses adam optimizer with an alpha of 0.02819, learning rate of 0.0020, batch size of 128, and three hidden layers (60, 60, 60). Both MLPs use ReLU activations for all hidden layers and output a continuous value representing the predicted GTEPS. Networks are trained for a maximum of 1000 training iterations with early stopping.

Once the model selection is completed, all models are evaluated using two performance statistics calculated from the predictions on a separate test set (not seen during training):

a) Percentage of Optimal: Represents the attained fraction of optimal performance achieved by the predicted configuration for each graph, given by $pred/opt$, where $pred$ is the actual GTEPS for the predicted configuration, and opt is the best possible GTEPS. opt is discovered via an exhaustive exploration of the entire optimization space. The reported performance is the geometric mean of all individual performances of graphs in the test dataset.

b) Speedup over Default Baseline (Galois): Represents the speedup of the predicted configuration over Galois. This baseline uses [data placement=device, thread block size=256, and registers allocation=moderate]. The formula is given by $pred/base$, where $base$ is the actual GTEPS for Galois. The geometric mean over all graphs is reported.

Particularly for the MLP, since we are able to produce predictions for multiple configurations, we include *top-1* and *top-3* performance measures for all metrics in Tables III-IV.

TABLE III: Percentage of optimal. Geometric mean is reported for all graph types. Combined represents the entire test set.

Model	Bipartite	Directed	Undirected	Combined
Top-1 MLP BFS	0.9587	0.9223	0.9688	0.9436
Top-3 MLP BFS	0.9788	0.9389	1.0000	**0.9633**
Top Config BFS	0.9893	0.8370	0.6468	0.8751
Top-1 MLP SSSP	0.9835	0.9703	0.9751	0.9766
Top-3 MLP SSSP	0.9944	0.9923	0.9986	**0.9940**
Top Config SSSP	0.9584	0.9081	0.9558	0.9355

TABLE IV: Speedup over default baseline (Galois). Geometric mean is reported for all graph types. Combined represents the entire test set.

Model	Bipartite	Directed	Undirected	Combined
Top-1 MLP BFS	1.9865	1.8630	2.1689	1.9503
Top-3 MLP BFS	2.0282	1.8966	2.2387	**1.9911**
Top Config BFS	2.0499	1.6906	1.4480	1.8089
Top-1 MLP SSSP	1.7695	1.7610	1.8759	1.7775
Top-3 MLP SSSP	1.7891	1.8010	1.9211	**1.8090**
Top Config SSSP	1.7244	1.6481	1.8387	1.7026

1) MLP: The percentage of optimal reported in Table III shows that the MLP is able to predict configurations close to the optimal performance. One clear advantage of the neural network approach is that we can easily predict GTEPS for all configurations. Given an input graph, we can feed the network with 30 pairs of graph attributes and configurations parameters, collecting the predicted GTEPS for all of them. The top k configurations are the ones with the highest k predicted values, and can be used to generate k kernels as part of the autotuning process. It is worth to note that in all cases the percentage of optimal is over 92% for BFS and over 97% for SSSP. As we increase the value of k, predictions become more accurate.

A summary of the number of times a configuration is chosen as optimal for all graphs in the test dataset is presented in Table V. Clearly, the optimal choice is input dependent and a one-size-fits-all heuristic, no matter how sophisticated, is bound to yield sub-optimal results.

TABLE V: Optimal Configuration Set for BFS and SSSP (Data Placement, Thread Block Size, Registers Allocation)

Configuration	Occurrences (BFS)	Occurrences (SSSP)
host,1024,aggressive	36	23
device,1024,aggressive	19	26
host,1024,moderate	5	9
host,512,moderate	4	0
host,512,aggressive	2	2
host,1024,conservative	1	0
host,512,conservative	1	0
host,64,conservative	1	0
host,64,moderate	1	0
device,1024,moderate	0	10

Using the information provided in Table V, we are able to compare the machine learning models directly with a "classifier" that always predicts the most frequent optimal configuration in the test dataset. The lines marked as "Top Config" in Tables III and IV show the geometric mean for such cases. The MLP outperforms this method by a good margin.

We performed additional experiments to evaluate the robustness of the MLPs. The idea is to observe the variance of the selected MLP architecture and hyperparameters, when trained using different 10 random weight initializations. The results are fairly robust for both graph algorithms, SSSP and BFS. In the case of BFS, there were only 3 instances were the trained models are showing higher variance. Complete results are shown in Fig. 6.

2) Decision Trees: The motivation behind training decision trees is that we could identify the most important features from the tree structure. After our experimentation with BFS, we discovered that for the bipartite graphs decision tree, the most important features, in order of importance, were *max_left_degree*, *left_size*, and *volume*. For the directed graphs, the most important features were *volume* and *outdegree_balanced_inequality_ratio*. For the undirected graphs, *volume*, *degree_assortativity*, *wedge_count*, and *gini_coefficient* (details of each attribute in [18]). For all decision trees, *volume* was an important attribute. For SSSP many of the same features emerged as critical for making

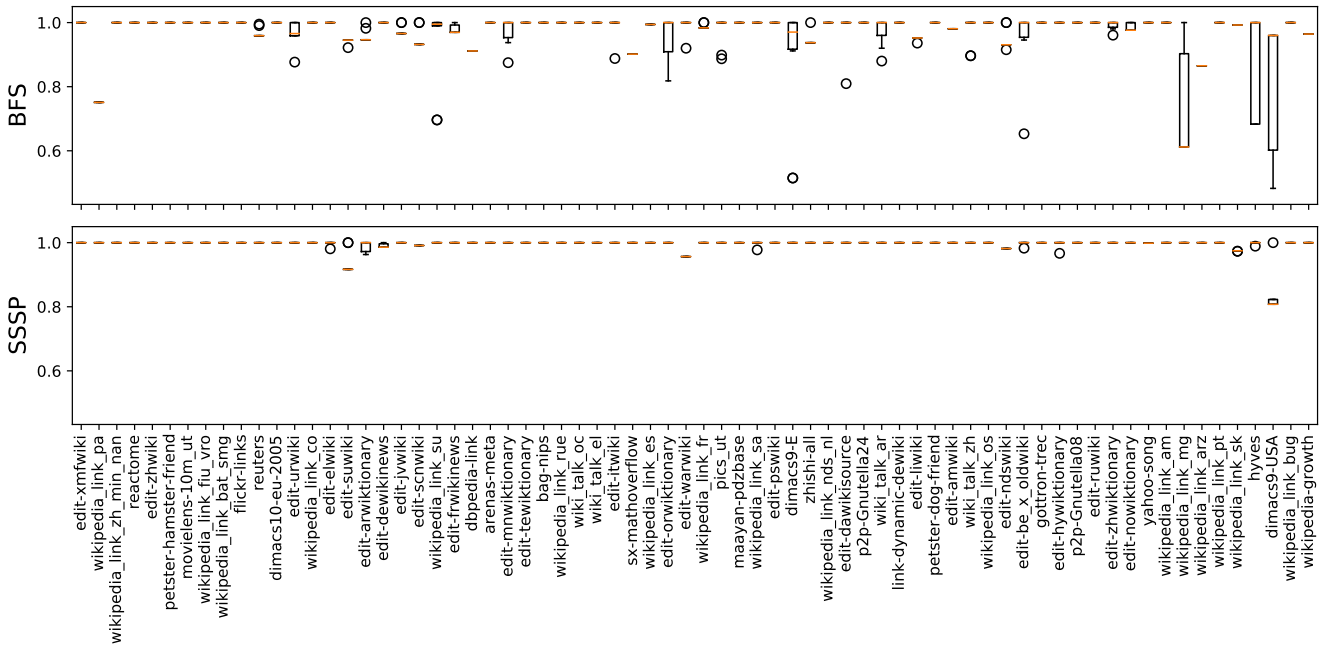

Fig. 6: Using the hyperparameters found by cross-validation, we train 10 different models with random weight initialization for each graph algorithm. The boxplots above summarize the percentage of optimal achieved by the top-3 predictions for all models applied to the test set. A low variance indicates higher confidence that the model is generalizing the problem rather than getting lucky.

optimization decisions. In addition, the graph *diameter* also proved consequential. Graphs with longer diameters, and consequently longer dependency chains, increase the register pressure. This increased pressure is amortized in the case of a full traversal, as in BFS, but is significant in the path discovery tasks embedded in SSSP. Therefore, unlike BFS, for many graph instances a moderate register allocation policy is necessary. This is reflected in the optimal configuration data presented in Table V.

VII. Related Work

A. Hybrid Graph Algorithms

There has been significant work on large-scale graph processing on CPU-GPU platforms in the past few years. Some have focused on algorithmic improvements [24], [25], while others have looked at developing specialized compiler- and system-level techniques [26]–[28]. More recently, researchers have proposed holistic frameworks for high-performance graph processing that combine optimization strategies at different levels [3], [4], [19]. Although these frameworks yield impressive speedups over traditional methods, it has been shown that the performance improvements are not consistent across different types of graphs [4], [21]. GraphIt aims to address this shortcoming with a DSL that enables programmers to select input-specific optimizations and algorithms [4]. GraphIt can consistently provide high performance across a range input graphs when the right parameters are selected. The responsibility of setting the parameters still lies with the programmer, however. This work aims to address this issue with a learned model that automatically selects the optimization parameters based on the characteristics of the input graph.

There have also been a few experimental studies that specifically look at input dependence in graph algorithms. The initial study by Burtscher *et al.* showed that it is possible to characterize irregularity along certain specific dimensions, such as memory coalescing [5]. Later studies build on this idea and look at other performance characteristics such as occupancy and task mapping [6], [21], [29]. However, none have explored input dependence in the context of collaborative design patterns, as this work does.

B. Machine Learning in Performance Modeling

The initial application of machine-learning-based performance tuning emerged as a response to prohibitively long tuning times for search-based autotuning [30]. Earlier work focused on pruning the optimization parameters space [31] and finding optimal compiler optimization sequences [32]. As neural networks and other learning algorithms gained popularity, they were applied to a variety of performance optimization problems. Many variants of popular ML techniques have been successfully applied to different branches – in performance optimization through code changes [33], [34], predicting runtime configurations [35], [36], selecting suitable data structures [37], identifying performance bottlenecks [38], [39], and system energy management [40], [41]. To the best of our knowledge this is the first attempt at using machine

learning to understand input sensitivity of graph algorithms under Unified Memory systems.

VIII. Conclusions

In this paper, we explore input-sensitive behavior of collaborative graph algorithms running on CPU-GPU systems that support Unified Memory. We provide contributions in identifying input-sensitive performance inefficiencies and evaluating machine learning approaches for modeling the relationship between input graph properties and performance.

Our experiments show that the selected graph attributes and performance criteria and metrics, are suitable for developing effective machine learning models. We achieve up to 96.33% of the maximum GTEPS performance for breadth-first search, and 99.40% for single source shortest path.

Future work includes generalizing the approach to a larger subset of important graph algorithms. In particular, we will investigate transfer learning methods to eliminate the need for retraining models on each new algorithm. We also expect to extend our training dataset by including graphs from the SNAP database and graphs generated synthetically.

Acknowledgement

This work was supported by the National Science Foundation through awards CNS-1253292 and OAC-1829644, and equipment grants by IBM and NVIDIA.

References

[1] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on gpus," in *Supercomputing (SC)*, 2015, pp. 1–12.

[2] A. McLaughlin and D. A. Bader, "Scalable and high performance betweenness centrality on the gpu," in *Supercomputing (SC)*, 2014, pp. 572–583.

[3] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *PACT16*, 2016.

[4] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A High-Performance Graph DSL," in *OOPSLA*, 2018.

[5] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *IISWC*, 2012, pp. 141–151.

[6] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *IISWC*, 2013, pp. 185–195.

[7] Y. Wu, Y. Wang, Y. Pan, C. Yang, and J. D. Owens, "Performance characterization of high-level programming models for gpu graph analytics," in *IISWC*, 2015, pp. 66–75.

[8] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[9] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.

[10] AMD, "Radeon Open Compute Manifest 3.7.0," https://https://rocm.github.io/, accessed: 2020-08-31.

[11] L. Li and B. M. Chapman, "Compiler assisted hybrid implicit and explicit GPU memory management under unified address space," in *Supercomputing (SC)*, 2019, pp. 51:1–51:16.

[12] T. Sultana, B. Allen, and A. Qasem, "Intelligent Data Placement on Discrete GPU Nodes with Unified Memory," in *PACT*, 2020, pp. 139—-151.

[13] J. Gomez-Luna, I. El Hajj, L.-W. Chang, V. Garcia-Flores, S. G. de Gonzalo, T. B. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: collaborative heterogeneous applications for integrated-architectures," in *ISPASS17*, 2017, pp. 43–54.

[14] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. R. Kaeli, "Hetero-mark, a benchmark suite for CPU-GPU collaborative computing," in *IISWC*, 2016, pp. 13–22.

[15] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, "Traversing large graphs on GPUs with unified memory," *VLDB*, vol. 13, no. 7, pp. 1119–1133, 2020.

[16] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W.-m. Hwu, "Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus," *arXiv preprint arXiv:2006.06890*, 2020.

[17] Y. Chen, I. B. Peng, Z. Peng, X. Liu, and B. Ren, "Atmem: Adaptive data placement in graph applications on heterogeneous memories," in *CGO*, 2020, p. 293–304.

[18] J. Kunegis, "Handbook of network analysis [konect – the koblenz network collection]," 2017.

[19] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *SOSP13*, 2013, p. 456–471.

[20] K. Zhou, M. Krentel, and J. Mellor-Crummey, "A tool for top-down performance analysis of gpu-accelerated applications," in *PPOPP20*, 2020, pp. 415–416.

[21] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *IISWC*, 2015, pp. 56–65.

[22] L. R. Group, "The llvm compiler infrastructure project," http://http://llvm.org/, 2007.

[23] D. Yan, W. Wang, and X. Chu, "Optimizing batched winograd convolution on gpus," in *PPoPP20*, 2020.

[24] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC12*, 2012, pp. 1–10.

[25] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *PPOPP12*, 2012.

[26] R. Kaleem, A. Venkat, S. Pai, M. Hall, and K. Pingali, "Synchronization trade-offs in gpu implementations of graph algorithms," in *IPDPS*, 2016, pp. 514–523.

[27] A. Li, W. Liu, L. Wang, K. Barker, and S. L. Song, "Warp-consolidation: A novel execution model for gpus," in *ICS*, 2018, pp. 53–64.

[28] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription," in *IPDPS*, 2020.

[29] M. Ahmad, H. Dogan, C. J. Michael, and O. Khan, "Heteromap: A runtime performance predictor for efficient processing of graph analytics on heterogeneous multi-accelerators," in *ISPASS*, 2019, pp. 268–281.

[30] P. Balaprakash, J. J. Dongarra, T. Gamblin, M. W. Hall, J. K. Hollingsworth, B. Norris, and R. W. Vuduc, "Autotuning in high-performance computing applications," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2068–2083, 2018.

[31] R. Vuduc, J. Demmel, and J. Bilmes, "Statistical models for empirical search-based performance tuning," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 65–94, 2004.

[32] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *CGO*, 2007.

[33] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Function merging by sequence alignment," in *CGO*, 2019.

[34] K. Stock, L.-N. Pouchet, and P. Sadayappan, "Using machine learning to improve automatic vectorization," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 50:1–50:23, Jan. 2012.

[35] S. Dublish, V. Nagarajan, and N. P. Topham, "Poise: Balancing Thread-Level Parallelism and Memory System Performance in GPUs Using Machine Learning," in *HPCA*, 2019, pp. 492–505.

[36] M. K. Emani and M. O' Boyle, "Celebrating diversity: A mixture of experts approach for runtime mapping in dynamic environments," in *PLDI*, 2015, pp. 499–508.

[37] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande, "Brainy: Effective Selection of Data Structures," in *PLDI*, 2011.

[38] J. J. Thiagarajan, R. Anirudh, B. Kailkhura, N. Jain, T. Islam, A. Bhatele, J. Yeom, and T. Gamblin, "Paddle: Performance analysis using a data-driven learning environment," in *IPDPS*, 2018.

[39] H. Xu, S. Wen, A. Gimenez, T. Gamblin, and X. Liu, "DR-BW: Identifying Bandwidth Contention in NUMA Architectures with Supervised Learning," in *IPDPS*, 2017.

[40] C. Imes, S. A. Hofmeyr, and H. Hoffmann, "Energy-efficient application resource scheduling using machine learning classifiers," in *ICPP*, 2018, pp. 45:1–45:11.

[41] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann, "CALOREE: learning control for predictable latency and low energy," in *ASPLOS*, 2018, pp. 184–198.