A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs

Ke Meng[1,2], Jiajia Li[3], Guangming Tan[1,2], Ninghui Sun[1,2]
[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
[3]Pacific Northwest National Laboratory
mengke@ncic.ac.cn Jiajia.Li@pnnl.gov {tgm, snh}@ict.ac.cn

Abstract

This paper proposes GSWITCH, a pattern-based algorithmic auto-tuning system that dynamically switches between optimization variants with negligible overhead. Its novelty lies in a small set of algorithmic patterns that allow for the configurable assembly of variants of the algorithm. The fast transition of GSWITCH is based on a machine learning model trained using 644 real graphs. Moreover, GSWITCH provides a simple programming interface that conceals low-level tuning details from the user. We evaluate GSWITCH on typical graph algorithms (BFS, CC, PR, SSSP, and BC) using Nvidia Kepler and Pascal GPUs. The results show that GSWITCH runs up to 10× faster than the best configuration of the state-of-the-art programmable GPU-based graph processing libraries on 10 representative graphs. GSWITCH outperforms Gunrock on 92.4% cases of 644 graphs which is the largest dataset evaluation reported to date.

CCS Concepts • **Software and its engineering → Software libraries and repositories**;

Keywords GPU, Graph processing, Auto-tuning

1 Introduction

Graph algorithms support a broad spectrum of applications. For example, social network analytics, such as PageRank [11] and HITS [54], rely heavily on graph models to represent relationships. Analysis tasks such as bug finding and network routing have used graph-based algorithms to improve their performance and accuracy [15, 57]. However, graph processing, especially for large-scale graphs, is computationally expensive. As GPUs provide higher parallelism and memory bandwidth than traditional CPUs, they are a promising hardware for accelerating graph applications. Some programmable graph processing frameworks have been developed on GPUs, such as CuSha [27], Frog [50], WS-VR [26], and Gunrock [58]. They integrate graph primitives and tend to provide a general-purpose library.

However, from the perspective of optimization, these frameworks often fail owing to the inherent properties of *input sensitivity* and *algorithmic diversity*, which are two fundamental problems in the performance tuning of many applications. (i) In the scenario featuring input sensitivity, dynamic graphs and the influence of diverse graph topologies are rarely considered in current graph algorithm libraries. A single optimization variant is typically used during the entire course of execution, which leads to significant bias in the best performance attained. For an instance of breadth-first search (BFS), the Gunrock library [58], the best hand-tuned implementation to date, achieves up to 47GTEPS on a scale-free graph and retards to 44MTEPs on a road-net graph. (ii) In the scenario featuring algorithmic diversity, an algorithm-specific optimization does not always work well for all cases. For example, WS-VR [26] and CuSha [27] perform well on PageRank-like algorithms that require dense workloads but show poor performance on traversal-based algorithms like SSSP with sparse workloads. On the contrary, direction optimization [7], which is applicable to a greater variety of graph algorithms, is used and tuned for only BFS in Gunrock. *This isolated performance tuning leads to difficulties in exploring all potential features and fails to reuse tuning strategies among different graph algorithms and datasets*.

Several studies have recently developed new auto-tuning techniques [5, 14, 32, 38] targeting the input sensitivity for performance critical applications. In contrast to traditional auto-tuning systems [16, 60] that generate libraries in an off-line way, an input sensitivity auto-tuning system requires an on-line manner of determining the best configuration or assembly of algorithms, referred to as *algorithmic auto-tuning*. The key is to specify an algorithm-choosing strategy that extracts cost-effective input features and maps them to a performance space, as in Ding et al.'s work [14]. Our contribution here is a set of algorithmic patterns that helps build an efficient algorithmic auto-tuner. This work focuses on GPU architectures because of their powerful capability in terms of both parallel computation and memory bandwidth.

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295716

We present GSWITCH [1]. It abstracts from a graph algorithm as a small set of algorithmic patterns that are characterized by quantitative parameters (features). Given different graph inputs, a graph algorithm is dynamically composed of concrete implementations of these algorithmic patterns during runtime. GSWITCH leverages a machine learning model to select good algorithmic configurations. Specifically, our main contributions are as follows:

- We define two properties, generality and significance, to identify good algorithmic patterns. Through a comprehensive study of graph algorithms using more than 1,200 real graphs, we extract five composable patterns satisfying these properties.
- We propose a machine learning based algorithmic auto-tuner called GSWITCH to dynamically generate high-performance graph algorithms. GSWITCH dynamically chooses the best choice of an algorithm with minimal overhead and no user intervention.
- GSWITCH is highly productive in terms of user programming. It consists of a front-end exposed to users and a transparent back-end. The front-end provides a succinct yet flexible abstraction for programming a graph processing algorithm. The back-end integrates individual highly optimized kernels into a parameterized kernel library.
- GSWITCH was implemented and evaluated on GPUs. Experiments on typical graph applications show that it outperforms the state-of-the-art libraries/frameworks by a factor of $2 \sim 10$. It achieves better performance than the highly hand-tuned counterpart Gunrock library [58] for 92.4% instances of 644 graphs.

To the best of our knowledge, this work is the first study to systematically investigate tuning variants of graph algorithms and develop an auto-tuning system for them on GPUs.

We have organized the remainder of this paper as follows: Section 2 provides related definitions and our motivations. Section 3 explores five algorithmic patterns, and Section 4 outlines the framework of GSWITCH and shows details of its implementation. In Section 5, we evaluate our system and analyze the effectiveness of our auto-tuning method. Section 6 surveys graph processing work on GPUs, and Section 7 concludes this work.

2 Background and Motivation

In this section, we introduce basic definitions for the graph processing framework. We also provide our motivations to give an intuition of why a pattern-based auto-tuner is both easy to implement algorithms and provides higher performance than state-of-the-art hand-tuned systems.

[1] GSWITCH is currently available in an open-source repository at https://github.com/PAA-NCIC/gswitch.

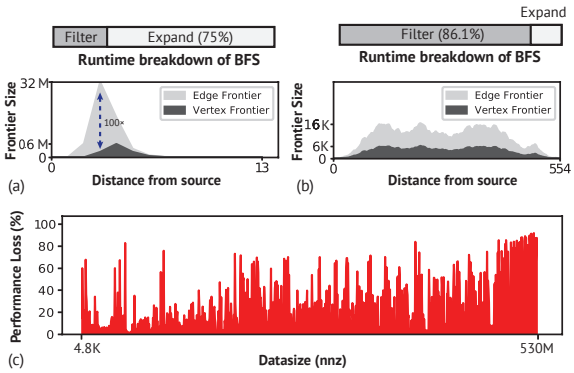

Figure 1. A motivating example of the BFS: The frontier expansion on (a) a scale-free graph `com-youtube` and (b) a road-net graph `roadNet-CA`; (c) Performance loss if we only use the *Push* [9] optimization variant on 1288 graphs.

2.1 Graph Algorithms

This work considers BSP (Bulk Synchronous Parallel) graph algorithms using the widely adopted *neighbor-based* property [62], where all vertices communicate only with their neighbors. A graph algorithm in the BSP model is treated as a sequence of super-steps that iteratively converge, with each step requiring global synchronization. Only a subset of the vertices or edges, known as the *active set*, participates in the computation in each iteration. An algorithm terminates with no more active elements. GSWITCH standardizes a graph algorithm as a quadratic-work process: The filtering step (referred as *Filter* in GSWITCH) generates the active set and updates their private data; then the expansion step (referred as *Expand* in GSWITCH) processes all the neighbors of the generated active set. Our *Filter-Expand* abstraction is inspired by the Gather-Apply-Scatter approach in PowerGraph [19] and the Advance-Filter-Update mechanism in Gunrock [58]. The main difference is that we simplify the abstraction by integrating the "Apply/Update" step into our *Filter*. In this paper, we use five typical graph applications to benchmark GSWITCH: Breadth-First Search (BFS) [7], Connected Components (CC) [53], PageRank (PR) [11], Single Source Shortest Path (SSSP) [42], and Betweenness Centrality (BC) [47] algorithms.

2.2 Motivation

Such challenges of graph processing on GPUs as irregular memory access patterns and unbalanced workloads have led to numerous optimization studies [7, 9, 44, 52, 58, 61, 66]. The idea of the algorithmic pattern is partially inspired by these efforts.

For example, directional optimization [7] has been widely used in some frameworks/libraries to handle irregular workloads. Figure 1 shows a breakdown of the runtime of the BFS, and the behavior at the edge and vertex frontiers of two types

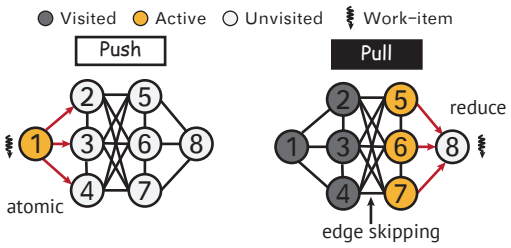

Figure 2. Direction: Push vs. Pull for BFS.

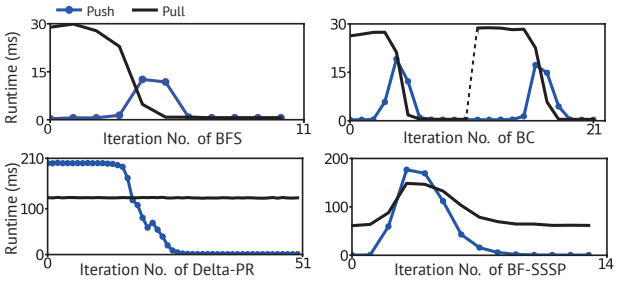

Figure 3. Runtime per-iteration in push and pull modes for graph `hollywood-09`.

of graphs: a scale-free graph `com-youtube` and a road-net graph `roadNet-CA`. The graph `com-youtube` (Figure 1 (a)) has a small diameter (13 from the x-axis) and workload explosion, its edges to be processed in middle iterations can be 100× more than the vertices in number. The *Expand* is its performance bottleneck (75% *Expand* vs. 25% *Filter*). By contrast, the graph `roadNet-CA` (Figure 1 (b)) has a larger diameter yet edges to process in each iteration. Thus, it spends more time on the *Filter* than the *Expand* (86.1% vs. 13.9%). If a single optimization variant (e.g., push [9]) is used on the two graphs, the loss of performance of the BFS can be as high as 80% (Figure 1 (c)). Some past work [7, 9, 21, 44, 52, 61, 66] has reported this performance variance and proposed specific tuning algorithms. We further abstract these characteristics as algorithmic patterns which heavily affect performance.

As we know, many influential performance-tuning strategies are available for implementing high-performance graph algorithms. Examples are strategies used to handle the load-imbalance within and between each warp/block, atomic-free operations, kernel fusion, coalesced memory access, among others [7, 9, 21, 22, 34, 35, 41, 44, 52, 61, 63, 66]. We advocate that a group of low-level optimizations should be considered as a single algorithmic pattern that can reduce the tuning space significantly. This motivates us to explore more algorithmic patterns and use them to build our algorithmic auto-tuner.

3 Exploring Algorithmic Patterns

A *pattern* here means a group of methods, strategies, or techniques to optimize the similar algorithmic paradigms in different graph applications, such as load-balance strategies, data structure. We call the optimization variants of a pattern as *candidates*. Specifically, we claim that an algorithmic pattern should have two properties:

- *Generality*: The pattern is effective for more than one application.
- *Significance*: The behavior of candidates is significantly different, depending on inclusion of the pattern.

The pattern set of GSWITCH includes *direction, the active set format, load balancing, stepping, and kernel fusion*. We describe each pattern from the perspective of their two properties below.

Pattern 1: Direction (Candidates: *Push* and *Pull*).

Generality: Direction is the push-pull dichotomy generalized form [7]. Figure 2 plots an example of the BFS. The push mode touches the neighboring edges of active vertices and updates a vertex property via an atomic operation. The pull mode touches the neighboring edges of inactive vertices and updates a vertex property via a reduction operation. The push-pull dichotomy can be applied to all the five graph algorithms as demonstrated in previous work [9, 52, 66]. In contrast to Gunrock, which only enables this optimization for BFS, we generalize this dichotomy as a common optimization for all the graph algorithms.

Significance: Figure 3 shows that the difference in performance between the push and pull modes is significant. For BFS and BC, the pull mode skips a large number of edges in the middle iterations. For Delta-PageRank [19], the push mode outperforms the pull mode when the active vertices decrease in length. For BF [2], the pull mode is faster than the push mode in some iterations with heavy workloads. Based on our observations of thousands of cases, we conclude that the pull mode is preferable in the middle iterations when the number of the active edges is greater than that of inactive edges. Therefore, using the numbers (and ratios) of the active and inactive elements in an iteration as tuning parameters is beneficial for auto-tuning.

Pattern 2: Active-set data-structure (Candidates: *Bitmap*, *Unsorted queue*, and *Sorted queue*).

Generality: As introduced in Section 2, the active set is a common data representation for most graph algorithms. GSWITCH chooses the data structure between the bitmap and the queue to maintain the active set [44]. Figure 4 shows these data structures for an example active vertex set ({2,3} for the second-level iteration) and their generations. The bitmap uses one bit for each vertex, where the value one represents an active vertex and zero represents an inactive vertex. Active vertices can also be described by a compact sorted/unsorted queue, which is generated by performing a scan-based method

[2] In this paper, SSSP refers to our implementation of dynamic stepping , BF is short for the unordered Bellman-Ford algorithm, and Δ-stepping represents the traditional implementation of stepping with the static priority threshold.

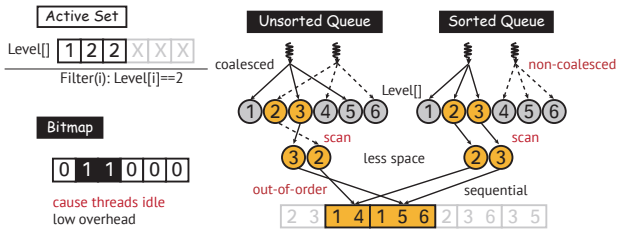

Figure 4. The second iteration of BFS with an active vertex set {2,3}. To select the active set in this iteration, the bitmap marks the two vertices as 1s and the rest as 0s, whereas the queue copies them to a compact in- or out-of-order array.

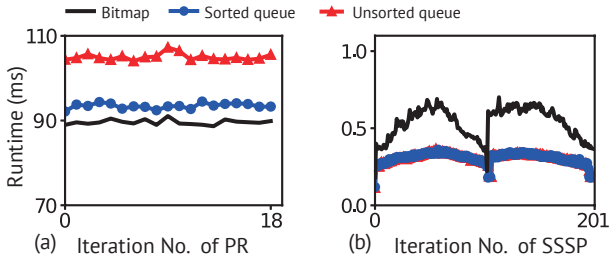

Figure 5. Runtime for each number of iterations in bitmap, unsorted queue, and sorted queue formats for (a) Pagerank in graph `kron_g500-log21` and (b) SSSP for graph `msdoor`.

that computes the offset of each thread. Comparing these two formats, the bitmap has no scan overhead but threads may be idle when assigned an inactive element, especially for small active sets. On the contrary, the queue does not cause idle threads but its generation step may be costly, especially for large active sets. The queue format can be either unsorted or sorted as shown in Figure 4. The generation of the unsorted queue is fast because it ensures coalesced memory access. The generation of the sorted queue is unfriendly to memory access, but the following Expand step benefits from the potentially contiguous memory access of the queue [33].

Significance: The performance of the tested candidates is shown in Figure 5. For an algorithm with a dense workload, like PageRank, all the vertices and edges are active in each iteration. Thus, the bitmap can avoid overhead due to the enqueue. For an algorithm with a sparse workload, like SSSP, compressing the sporadic active vertices into the queue is useful. In conclusion, bitmap is more friendly to dense workloads, the unsorted queue is more friendly to sparse workloads, and the sorted queue is in the middle. Tuning parameters such as the active set size and the number of corresponding edges are critical to this pattern.

Pattern 3: Load balance (Candidates: *TWC*, *WM*, *CM*, and *STRICT*).

Generality: Due to the mismatch between the irregular computations in graph algorithms and the Single Instruction Multiple Threads (SIMT) architecture of GPUs, this pattern is important for GPU optimizations. Figure 6 illustrates four load-balancing strategies:

- The Thread, Warp, and CTA (*TWC*) method was first introduced by the B40C [41]. For Nvidia GPUs, a warp is a set of threads working in a lockstep fashion, while CTA (Cooperative Thread Array) is a set of warps. The active vertex set is divided into low-, medium-, and high-degreed vertices according to their out-degrees, which are mapped to a thread, a thread warp, and a CTA separately.
- The Warp Mapping (*WM*) method redresses the imbalance in a warp. A warp processes continuous vertices from the active set as a batch. It loads the *warpsize* of their neighbors into GPU shared memory each time, until all their neighbors have been processed. A binary search on the loaded edges is performed to find the sources of the neighbors. Finally, the results are written back to global memory.
- The CTA Mapping (*CM*) method is similar to WM, and deals with the balance in a CTA. A CTA loads continuous *blocksize* vertices as a batch and their *blocksize* neighbors into the shared memory until all neighbors have been processed. A larger binary search is used to compute the source of the loaded neighbors, and synchronization is needed for threads in a CTA.
- The *STRICT* method handles the balance both in CTAs and across them. It is modified from the load-balance partitioning (LB) in [13]. This method forces each CTA to maintain an equal number of vertices and edges, by using a sorted search to find the optimal partitioning of the workload. Then, the edge and vertex lists are then both partitioned.

Significance: *STRICT* obtains the best load balance but its overhead is the highest, *WM/CM* is in between, and *TWC* has the lowest overhead but it is not balanced well. The main overhead of WM is the binary search, which can be completed in no more than $\log_2(warpsize)$ steps, while that of CM is $\log_2(blocksize)$ steps. As shown in Figure 7, *TWC* runs faster owing to its lower overhead, and *CM* and *WM* perform better with irregular workloads. When the workload contains a hub vertex which is connected to a large number of neighboring vertices, *STRICT* is the best method.

Pattern 4: Stepping (Candidates: *Increase*, *Decrease*, and *Remain*).

Generality: This pattern is also related to the active set as it specifies its execution behavior for different graph processing algorithms. A monotonic algorithm, such as SSSP, can be implemented in either ordered or unordered mode [21]. An ordered implementation updates only high-priority elements of the active set (e.g., the Δ-stepping algorithm [42]) to avoid

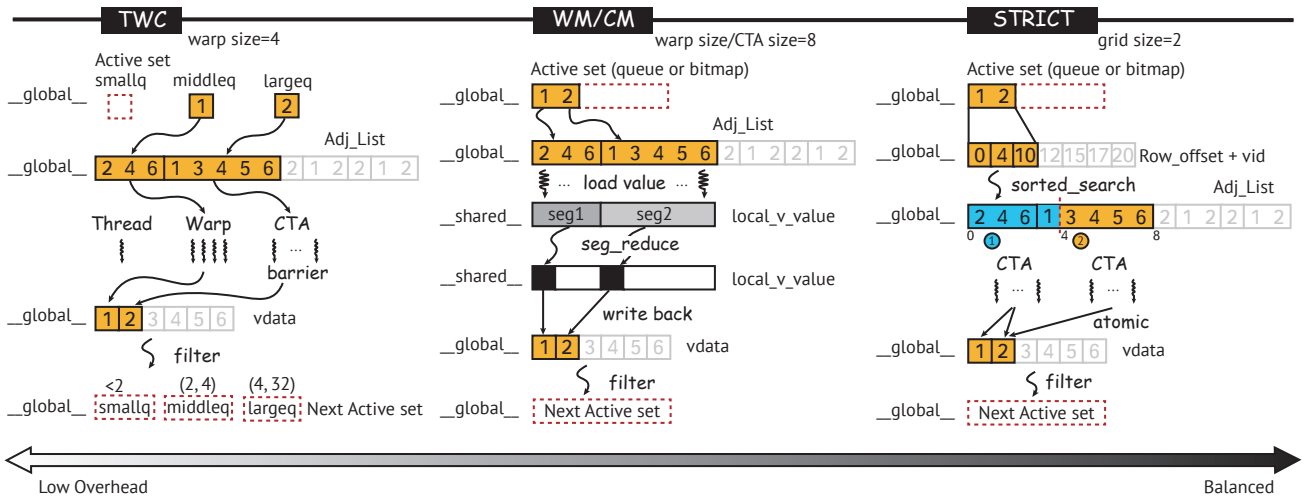

Figure 6. Load-balancing strategies: TWC, WM, CM, and STRICT.

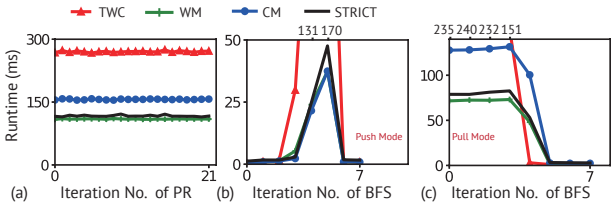

Figure 7. Runtime per iteration on TWC, WM, CM, and STRICT methods for Load-Balancing. (a) The PageRank for `soc-orkut`. (b) The push mode of BFS for `soc-orkut`. (c) The pull mode of BFS for `soc-orkut`.

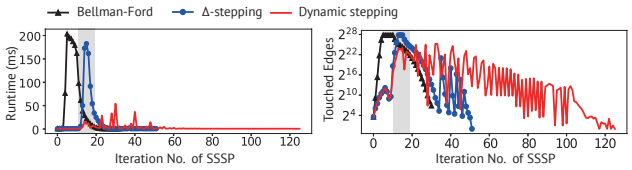

Figure 8. A comparsion among the unordered Bellman-Ford, the ordered Δ-stepping , and dynamic stepping for SSSP on graph `soc-orkut`.

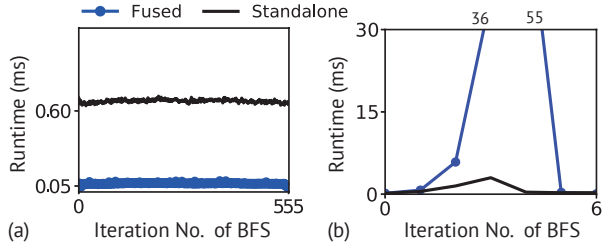

Figure 9. Runtime per iteration runtime of variants with fused or standalone candidate for graphs (a) `roadNet-CA` and (b) `soc-orkut`.

unnecessary computation. An unordered implementation engages as many active elements as possible to pursue high parallelism (e.g., the Bellman-Ford algorithm). With online algorithmic auto-tuning in mind, we propose a dynamic priority threshold and amend it by comparing the (estimated) number of edges between the previous iterations and the ones following. If the number of edges increases or decreases by some margin, we increase or decrease the priority threshold accordingly by a pre-set step size; otherwise, the priority threshold stays constant. From our experiments, the increase/decrease ratio was set to 35%.

Significance: Figure 8 shows the difference among the unordered version, the static ordered version (Δ-stepping), and the dynamic ordered version. We set the priority threshold for the static unordered version to the cw/d, used in [13]. The dynamic ordered version was more flexible when facing workload explosion. To determine whether to increase or decrease the stepping threshold, GSWITCH collects the characteristics of the current workload (the number of edges to be explored and the edge distribution) as tuning parameters.

Pattern 5: Kernel fusion (Candidates: *Standalone* and *Fused*).

Generality: Kernel fusion fuses multiple operations into a single kernel to improve the operational intensity. To avoid processing duplicated vertices, active set generation and processing are usually implemented as two separate kernels such as in Gunrock and Enterprise. A duplicate removal process is required for them. However, for some sparse graphs with stable workloads, if the attributes of the dataset and historical information imply a shortcut, we use the kernel fusion to speed up the execution. Duplication-tolerant techniques such as bitmap marking is applied to guarantee the correctness in a kernel-fusion variant. If the runtime of the last iteration is far

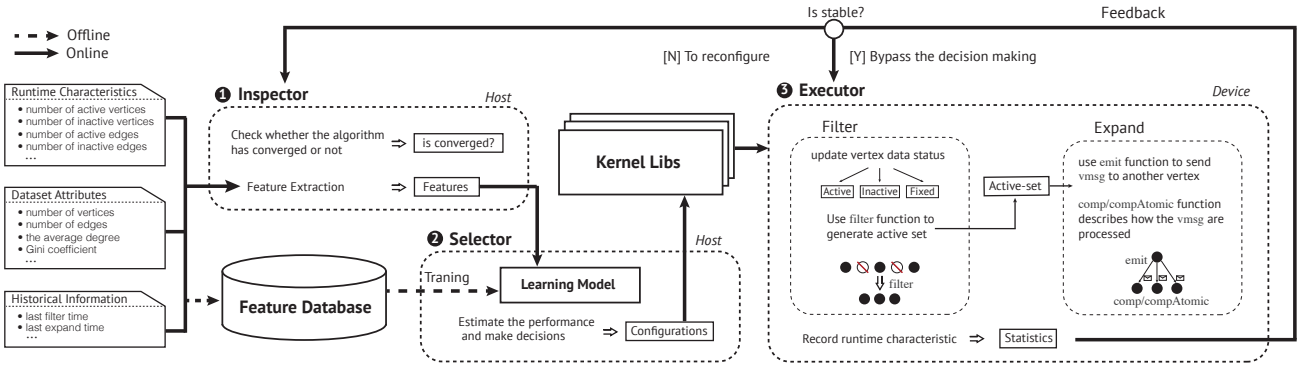

Figure 10. The overview of GSWITCH. On the host side, the Inspector extracts the feature vectors that are fed to the learning model in the Selector. On the device side, the Executor processes the selected kernels and generates the feedback for the next iteration.

Table 1. Feature vector in GSWITCH.

Parameter	Meaning	Correlation
Dataset attributes		
N, M	the number of vertices and edges	P3,5
d	the average of degrees	P3,5
σ_d	the standard deviation of degrees	P3,5
r_d	the relative range of degrees	P3,5
GI	Gini coefficient [29]	P3,5
H_{er}	Relative edge distribution entropy [29]	P3,5
Runtime Characteristics		
V_a, E_a	the number of active vertices and edges	P1,2,3,4
V_{ia}, E_{ia}	the number of inactive vertices and edegs	P1,2,3,4
V_{ap}, E_{ap}	the ratio of active vertices and edges	P1,2,3,4
V_{iap}, E_{iap}	the ratio of inactive vertices and edges	P1,2,3,4
cd	the average of degrees in current workload	P1,2,3,4
r_{cd}	the relative range of degrees in current workload	P1,2,3,4
Historical Information		
t_f	the time of the last Filter step	P5
t_e	the time of the last Expand step	P5
T_f	the average time of previous Filter steps	P5
T_e	the average time of previous Expand steps	P5

longer than the average runtime in the fused mode, GSWITCH switches back to the standalone mode.

Significance: As illustrated in Figure 9 (a), the fused version runs 12× faster than the standalone one, which improves performance significantly on road-net graphs. However, for dense inputs like those from social graphs, where the workload is irregular and unpredictable, too many duplicates are generated from fused version to achieve reasonable performance (Figure 9 (b)). Features of the input graph and the runtime of previous iterations are relevant tuning parameters for this pattern.

The parameters of these patterns are summarized in Table 1. We refer to Pattern i as Pi for short. By determining a proper configuration of these parameterized patterns, GSWITCH obtains an optimized execution of a given graph processing algorithm. The machine learning technique used to train these candidate values will be discussed in the next section.

4 Implementation of GSWITCH

4.1 GSWITCH Overview

As the algorithmic optimizations are abstracted as assembled parameterized patterns, it is natural to adopt a machine learning based method, as they have been successfully used in recent auto-tuners [12, 32, 48, 64]. As shown in Figure 10, in every iteration (or super-step) GSWITCH selects optimized variants on-the-fly through the **Inspector**, **Selector**, and **Executor** stages. Two knowledge databases (feature database and parameterized kernel library) are constructed to train the model offline.

- **Inspector**. The inspector (labeled as ❶) first checks for the convergence of the previous iteration. If the previous iteration has not converged, it starts a new iteration, collects the values of the pre-defined features: dataset attributes, runtime characteristics, and historical information, and saves them as a vector. These feature values are the inputs to the selector.
- **Selector**. The selector (labeled as ❷) is a black-box learning model which is trained offline from the feature database. It takes the feature vector from the Inspector to predict the optimal kernel configuration for the next iteration. The configuration is saved as an entry in the kernel library.
- **Executor**. The executor (labeled as ❸) chooses the determined kernels from the kernel library and executes them appropriately on the GPU. Meanwhile, some runtime characteristics are profiled for use as feedback for the next inspector.

As shown in Figure 10, the inspector and selector stages are executed on the host (CPUs), while the executor is processed

on the device (GPUs). The kernel library stores the implementation variants of the Filter and Expand primitives. At the end of each iteration, a feedback stage copies the runtime characteristics and historical information of the last iteration from the device to the host.

4.2 Programming APIs

To implement the customized graph application, users need to provide four functions for GSWITCH to implement their graph applications: `filter`, `emit`, `comp` and `compAtomic`. Figure 11 gives an example of these functions that together implement BFS. The `filter` function provides a predicate to refine the active set for the current iteration. The `emit` function defines the message that the source vertex should send. The `comp` and `compAtomic` functions both describe how the message is processed in the receiving vertex, with the former being atomic-free while the latter is not. No tuning parameters are required and all the tuning details are opaque to users.

```
__device__ Status filter(int vid, G g){
  int lvl = data_of(vid);
  if(lvl == g.level()){return Active;}
  else if(lvl < 0) return Inactive;
  else return Fixed;
}
__device__ int emit(int vid, Empty* weight, G g){
  return data_of(vid)+1;
}
__device__ void compAtomic(int* u, int lvl, G g){
  atomicExch(u, lvl);
}
__device__ void comp(int* u, int lvl, G g){
  *u = lvl;
}
```

Figure 11. An example of BFS algorithm in GSWITCH.

4.3 Feature Extraction

The feature vector for training and inference on the model is listed in Table 1. These features are easily collected even during runtime because most of the required runtime characteristics are side products of the Filter and Expand steps. These features fall in three dimensions:

- The dataset attributes provide the basic description of the graph. First, two parameters are used to represent the basic structure of the graph: N (the number of vertices) and M (the number of edges). Second, d, σ_d, and r_d are added to describe the distribution of the edges. To classify whether a graph is regular or irregular, we introduce the Gini coefficient and the relative edge distribution entropy to measure the equality of degree distributions [29]. The attributes of the dataset are computed only once while loading the data into the memory.
- The runtime characteristics describe the workload in each iteration. The numbers and ratios of active and inactive elements in the graph help determine the direction (P1). Once this is done, GSWITCH chooses the active or inactive elements as the current workload. cd and r_{cd} describe the degree distribution of the given workload. The active set format (P2) and load-balancing strategies (P3) are highly relevant to the features of the given workload.
- Historical information reveals the workload imbalance of previous iterations. By comparing the time taken for the last iteration with the average time for all previous iterations, we can estimate the workload of the last iteration without copying the runtime characteristics from the device.

The features are selected based on a statistical analysis. Figure 12 shows six most prominent features for the five patterns on 644 graphs [1], where the distribution of each pattern reflects the influence of the features. Figure 12 (a) shows that the pull mode improvements when the number of inactive edges is small; By contrast, the push mode is preferred when the number of inactive edges increases. Figure 12 (b) shows that the queue mode performs well when the number of active vertices is small. Figure 12 (c) and (d) show that the STRICT mode is beneficial when the data are irregular and the number of active vertices is large. Figure 12 (e) indicates that the stepping threshold should be increased when the number of active edges is small. Figure 12 (f) shows the *Gini* coefficient that describes the inequality of the degree distribution, from which we can conclude that fused kernels are preferred when the vertices in a graph have similar numbers of neighbors.

4.4 Model Generation

We treat patterns as independent decision goals, and the statistics of each iteration constitute a record in the feature database. For example, one record of the graph `web-it-2004` is R ={509338, 7178410, 28.2, 58.7, 16.6, 0.55, 0.94, 289, 508987, 2694, 14353300, 0.0006, 0.9994, 0.0002, 0.9998, 9.32, 137.7, 1.2, 1.3, 0.4, 0.15}. We ran all the implementations of the kernel library on 644 graphs [1] for all the benchmarks and gathered a total of 386,780 records (one record for each iteration). The true optimal configurations were attained via brute-force experimentation.

Decision making can be regarded as a classification problem. GSWITCH builds a classifier for each pattern. The classification process can be formulated as $f(R) = OPT$. For example, the output OPT here in the directional classifier can be either push or pull. The CART (Classification and Regression Tree) algorithm is used to generate the decision tree. We choose it as our learning model for two main reasons. First, this model is easy to convert the resulting rules to if-else sentences, which is convenient for transplanting them to diverse platforms and integrating them with other runtime systems. Second, the model is easy to understand and interpret while its prediction overhead is low. However, the decision-tree

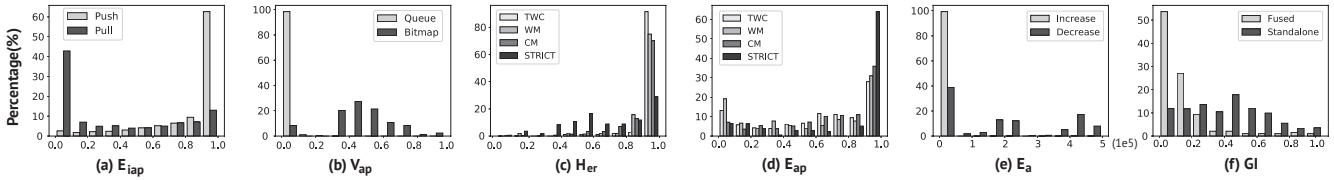

Figure 12. The distribution of the optimal strategy with different parameter values, where the y-axis shows the percentage of each parameter falling into its value intervals.

model is prone to overfitting. To improve the generality of the model, we tailor the generated decision tree and keep the its height as low as possible.

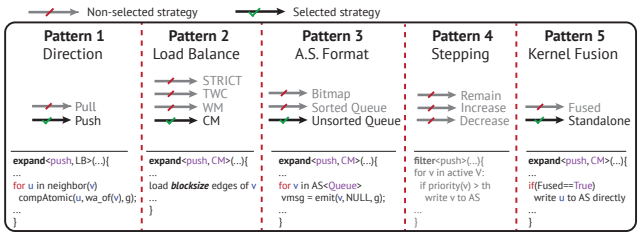

Figure 13. The kernel searching process of the selector along with some interpretable rules in the decision path of one iteration.

Gswitch Best

lvl	Push				Pull				Gswitch	Best
	TWC	WM	CM	STRICT	TWC	WM	CM	STRICT		
1	0.22	0.20	0.21	0.21	97.0	49.5	44.4	124	0.20	0.20
2	0.59	0.46	0.40	0.40	98.6	50.4	46.3	123	0.40	0.40
3	1.61	0.82	0.76	0.54	96.5	50.4	44.9	126	0.54	0.54
4	38.7	4.74	2.03	1.50	58.8	49.2	44.8	122	1.50	1.50
5	58.5	15.5	17.7	17.9	1.80	33.5	31.3	71.3	1.80	1.80
6	61.8	27.2	30.4	29.7	0.38	0.86	2.01	2.74	0.38	0.38
7	0.97	0.68	0.68	0.82	0.39	0.40	0.39	0.40	0.39	0.39
∑	162	49.7	52.2	51.1	354	234	214	569	5.21	5.21

Figure 14. The runtime of different strategies of BFS for graph `orkut`. "Gswitch" indicates the predicted strategies of GSWITCH, "Best" indicates the true optimal strategies.

4.5 Kernel Searching

In the low-level implementation, the parameterized kernels are described as C++ templates. Some patterns (e.g., P1: direction) are implemented as standalone kernels, while others (e.g., P2: active set formats) are implemented as a combination of branches and conditions. In total, we had 12 standalone kernels supporting 156 implementation variants. In each iteration, we needed to choose a filter primitive from 12 candidates and one expand primitive from 144 candidates.

We made decisions following a given order and each pattern had one classifier. Figure 13 illustrates the kernel searching process for one iteration of BFS. We first determine the direction, because the workloads in push and pull are completely different. Load-balancing strategies are then considered, followed by the active set format. We finally decide the stepping configuration as well as whether to enable the kernel fusion. Figure 14 shows the searching path of the graph `orkut` for the BFS algorithm along the dimensions of the direction and the load-balancing. In this case, GSWITCH chooses the optimal strategy in each iteration.

Table 2. Representative graphs for benchmarking.

Graphs	Vertices	Edges	Max Degree	Domain
soc-orkut	3M	212.7M	27,466	SN
soc-pokec	1.6M	61M	20,518	SN
web-uk-2005	129K	23M	850	WG
web-wikipedia-2009	1.8M	9M	2,623	WG
kron_g500-log21	2.1M	182.1M	213,904	GG
rgg_n_2_24	16.8M	265.1M	40	GG
roadNet-CA	1.9M	5.5M	121	RN
roadNet-TX	1.4M	3.8M	12	RN
sc-msdoor	415K	19.8M	76	SC
sc-ldoor	952K	42M	76	SC

Graphs domains are: SN: Social Network, WG: Web Graph, GG: Generated Graph, RN: Road Network, SC: Scientific Computing.

5 Evaluation

5.1 Experimental Setup

Platform. We ran all experiments on two systems, both on Linux servers with 2.10GHz E5–2620 v2 Intel Xeon CPUs and 48 GB of main memory. One had an Nvidia K40m GPU with 12 GB of global memory, and the other had an Nvidia P100 GPU with 16 GB of global memory. We compiled all the GPU programs using NVIDIA's `nvcc` compiler (version 7.5.17) and the `-O3` flag.

Dataset. We randomly chose 1,288 graphs from the network repository [1] [3]. Half of them (644) were used as the

[3] All graphs were transformed into undirected ones.

Table 3. The evaluation time of GSWITCH vs. other GPU graph processing libraries and hardcoded implementations.

		Runtime (ms) [lower is better]									
		Social Network		Web Graph		Generated Graph		Road Network		Scientific Computing	
Alg.	**Lib.**	**orkut**	**pokec**	**web-uk-05**	**web-wp-09**	**kron-21**	**rgg_n24**	**roadnNet-CA**	**roadNet-TX**	**msdoor**	**ldoor**
BFS	Enterprise	30.7	11.2	3.1	87	10	-	-	47	44	68
	Gunrock	6.1	12	1.6	25	4.2	606	104	82	42	64
	Gswitch	**5.5**	**3.8**	**1.2**	**18**	**2.9**	**344**	**39**	**47**	**23**	**33**
CC	GPUCC	**84**	**21**	5.7	14	128	-	9.1	6.7	6.3	13
	Gunrock	161	38	50	36	236	305	17	13	17	40
	Gswitch	111	35	**3.9**	**11**	**89**	**143**	**6.1**	**4.7**	**3.9**	**7.5**
PR	WS-VR	16567 (79)	2718 (51)	116 (27)	408 (49)	6600 (35)	7769 (67)	61 (52)	43 (51)	253 (60)	421 (58)
	Gunrock	4069 (22)	733 (21)	70 (22)	1420 (24)	3009 (19)	1533 (19)	190 (21)	134 (21)	54 (18)	107 (18)
	Gswitch	**2201 (22)**	**433 (21)**	**59 (22)**	**117 (24)**	**1759 (19)**	**1011 (19)**	**32 (20)**	**20 (20)**	**41 (18)**	**84 (18)**
SSSP	Frog	2424	534	35	436	847	-	**44**	**28**	199	363
	Gunrock	1096	385	29	112	248	99663	179	196	175	582
	Gswitch	**407**	**144**	**16**	**83**	**247**	**2772**	84	90	**46**	**83**
BC	GPUBC	286	46	7.6	100	**57**	1298	194	182	87	250
	Gunrock	427	92	11	88	371	1669	116	227	62	98
	Gswitch	**85**	**28**	**2.9**	**55**	91	**916**	**78**	**101**	**56**	**79**

The numbers in brackets are the numbers of iterations. As Gunrock updates its code continuously, some performance results we reproduced for it are inconsistent with the results of the original paper [58] (e.g., on `rgg_n24`) but consistent with the results of the authors' more recent paper [59].

training set for model generation, while the rest were used as the evaluation set. The evaluation set has no overlap with the training set. Ten-fold cross-validation was used to evaluate the accuracy of the model. As shown in Table 2, we chose 10 representative graphs from the evaluation set to analyze.

Baseline. GSWITCH was compared with several state-of-the-art programmable GPU-based graph processing frameworks/libraries. According to the latest results in [2, 59], Gunrock [58] delivered relatively better performance than the others owing to its continuous evolution. We compared our method with Gunrock on all test cases. For each graph algorithm, we selected a specialized, optimized implementation from other frameworks/libraries that beats Gunrock in some cases, which were Enterprise [33], GPUCC [53], WS-VR [26], Frog [50] and GPUBC [47] for BFS, CC, PR, SSSP and BC, respectively.

5.2 Overall Performance

Table 3 shows the performance comparison of the graph libraries as described in Section 5.1. For the BFS benchmark, Gunrock, GSWITCH, and the hand-tuned version of Enterprise enabled the direction-switching and idempotence optimization in BFS. However, Gunrock needed user-provided tuning parameters, such as *do_a* and *do_b* to decide when to switch. These parameters can vary significantly for different graphs. For example, Gunrock's BFS achieved its best performance on graph `soc-orkut` when $do\_a = 0.12$ and $do\_b = 0.1$, while on graph `roadNet-CA`, the best setting was $do\_a = 1$ and $do\_b = 10$. The Enterprise adopted static rule-based switching that led to suboptimal results in some cases, for instance, on graphs `soc-orkut` and `web-wikipedia-2009`. For the CC benchmark, our method beat all the programmable GPU graph libraries largely due to the transition of the active set format. GSWITCH was slower than GPUCC in some cases because the latter used specific optimizations, which can not be generalized. For the PageRank benchmark, each library had a different implementation but used the same terminal condition. WS-VR used the pull mode and the WM load-balancing strategy for all cases, whereas Gunrock used the push mode and the LB [13] load-balancing strategy for all cases. By contrast, GSWITCH used different strategies for different inputs. For the SSSP benchmark, our dynamic stepping optimization reduced the number of touched edges significantly compared to the static Δ-stepping version. Frog performed well on some graphs because it used an asynchronous algorithm that convergences more quickly. For the BC benchmark, both the GPUBC and Gunrock used a push-based implementation, while GSWITCH performed faster than Gunrock due to the generalized directional optimization.

Figure 15 shows the performance of GSWITCH compared with that of Gunrock on K40m (a) and P100 (b) on the evaluation set. On K40m, the results show that the average performance of GSWITCH was approximately 2.5~4.6× faster than Gunrock on average, and it achieved positive speedups in about 84%~96% of the cases. On P100, we retrained the model to test the portability of GSWITCH. The results show that GSWITCH was about 2~3.3× faster than Gunrock on average and 94%~99% of the cases were positive. GSWITCH can thus choose the suitable strategies automatically, while Gunrock relies on user-provided tuning parameters that are frequently unavailable in real-world situations.

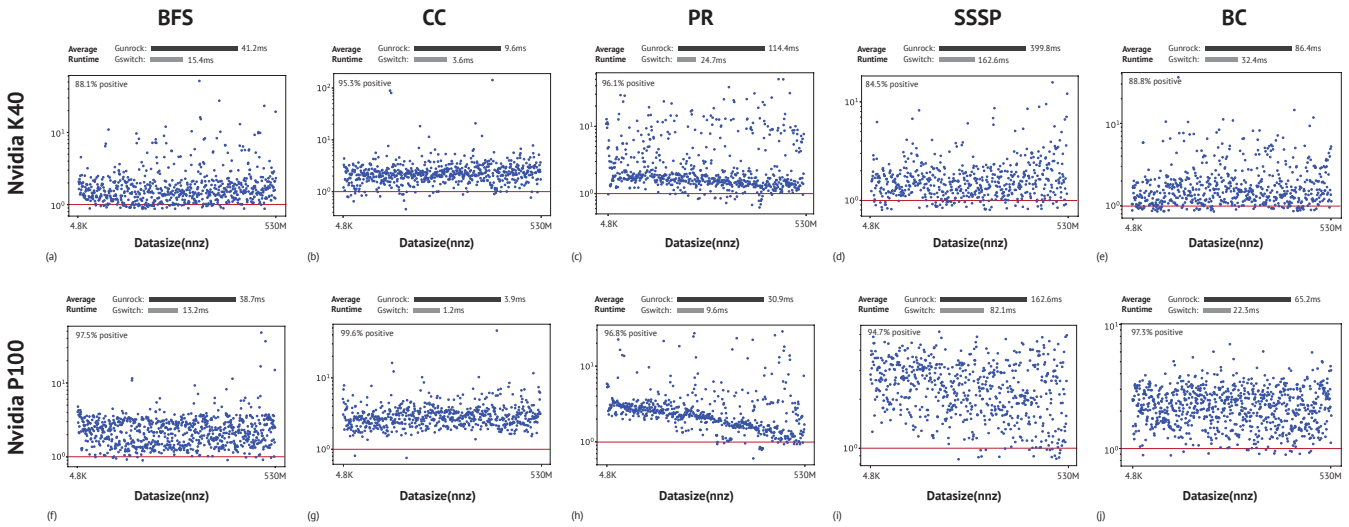

Figure 15. Performance improvement normalized to Gunrock on both K40m and P100 GPUs for all the five graph algorithms.

5.3 Effect of Individual Patterns

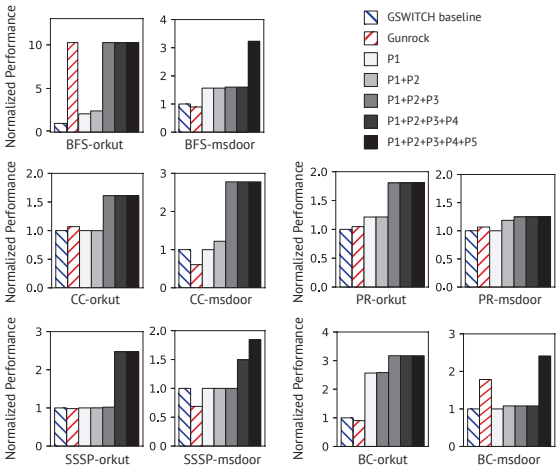

Figure 16. The incremental perfomance of GSWITCH compared with Gunrock.

To further understand the improvement in performance effected by each pattern, we illustrate the incremental speedups of a scale-free graph (`soc-orkut`) and a mesh-like graph (`msdoor`) in Figure 16. We used Gunrock as the static optimal version of each algorithm. The results show that the GSWITCH baseline (the non-switching version) delivers a similar performance to that of the Gunrock implementation. Therefore, the superiority of GSWITCH in performance over Gunrock is mainly owing to dynamic switching. Moreover, the importance of the different algorithmic patterns on performance improvement varied from algorithm to algorithm. For traversal-based algorithms such as BFS and BC, the directional optimization (P1) yielded an approximately 2× bump

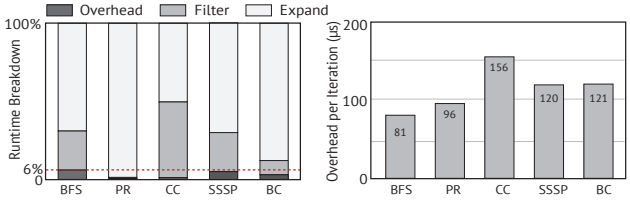

Figure 17. The normalized time breakdown of five benchmarks and the overhead of dynamical switching on `soc-orkut`.

in performance. The active set format (P2) brought about a 10% performance improvement on dense workloads. Load balancing (P3) is important for scale-free graphs, where the workload of each iteration varies significantly. Stepping optimization (P4) profiled in only monotonic algorithms, and the kernel fusion (P5) reduced the filter time, which is the bottleneck in mesh-like graphs.

5.4 Accuracy and Overhead

Each record contained 21 features and one label showing the optimal strategy as described in Section 3. GSWITCH had a corresponding classifier for each pattern to decide which strategy to choose. We statistically analyzed the accuracy of each pattern during the online decision making. The accuracies were 98%, 97%, 85%, 82% and 94% for the five patterns, respectively. It is interesting to note that GSWITCH achieved good performance even though the learning model could not correctly predict in some cases. More features or a more complex model can improve accuracy, but will also increase the runtime of feature extraction and training.

Figure 17 shows the runtime breakdown for the five benchmarks. For edge-centric algorithms such as CC, the active

set is large. Thus the Filter and Expand steps cost similar time. For applications such as PR and BC, which have more edges to process, the Expand step dominates. The overhead in GSWITCH includes collecting the runtime characteristics and making the decisions. As described in Section 4.3, the feature vector can be collected efficiently without extra computation, which cost 58∼120 μs in each iteration. The overhead of dynamic switching in GSWITCH was affordable, as it cost at most 6% of total runtime.

6 Related Work

GSWITCH was developed as an auto-tuner to implement adaptive graph algorithms. In contrast to traditional auto-tuners [16, 60], it derives from the emerging algorithmic auto-tuning approaches [5, 14, 32, 36]. For the optimization of graph processing algorithms, its design and implementation share with and are inspired from past work. We summarize related work to clarify the common features with other work and highlight GSWITCH's superiority in implementing a highly efficient graph algorithm library.

First, a large number of hardwired GPU implementations provide enough references for the kernel library of the GSWITCH auto-tuner. We refer them to variants of training and searching optimization on GPUs. For example, Merrill et al.'s BFS algorithm [41] proposed an adaptive load-balanced strategy to process a frontier with a thread, a warp, or a CTA, according to the number of neighbors of the given vertex. Enterprise [33] implements the direction-optimized BFS with streamlined GPU thread scheduling. Verstraaten et al. [56] implemented the directional optimization using a model-driven approach. Soman et al. [53] implemented the CC algorithm with *edge-centric* abstraction and accelerated the root finding procedure with the pointer jumping technique. Davidson et al.'s [13] SSSP algorithm uses a near-far approach to reduce the number of touched edges during the traversal, trading parallelism for work-efficiency. Mclaughlin and Bader [40] implemented an efficient BC algorithm to solve the load imbalance issue in threads. A straightforward combination of these optimization methods fails to achieve better performance. GSWITCH leverages a machine learning model to determine a proper configuration.

Second, several studies have discussed the trade-off among performance tuning strategies. Unfortunately, they are confined to only a specific graph algorithm. For example, Beamer et al.'s work [7] first introduced the directional optimization in an implementation of BFS on CPUs. Nasre et al. [44] analyzed the trade-offs between data-driven and topology-driven approaches. Hassaan et al.'s work [21] discussed the pros and cons of the ordered and unordered implementations of irregular algorithms and attempted to balance parallelism with work-efficiency. Inspired by these individual efforts, GSWITCH goes further to examine the correlation among multiple algorithms in terms of tuning. The key point is that our extracted algorithmic patterns have not been revealed in previous work.

Last, Gunrock [58] integrated the above-mentioned tuning strategies to implement a number of graph primitives. To some extent, this applies common tuning strategies to different primitives. However, the optimization depends on the users' priori knowledge. It is a static configuration that may become suboptimal. GSWITCH breaks the barrier by automatically building a better algorithm by assembling strategies at runtime owing to the algorithmic patterns.

7 Conclusion

In this work, we proposed GSWITCH, an adaptive performance tuning system for graph processing algorithms. Its novelty is in a set of algorithmic patterns that naturally enable the adoption of a machine learning model. The results of evaluations show that GSWITCH yields better performance than the best configurations of four other state-of-the-art GPU graph processing libraries. This work showcases algorithmic auto-tuning techniques. We advocate that more research effort should be put into advancing algorithmic auto-tuning, especially for input-sensitive workloads such as graph and sparse tensor algorithms. However, our patterns are not fully complete yet. We believe that many opportunities and open research questions persist for extracting additional patterns.

8 Acknowledgments

We thank all reviewers for constructive comments that helped us further clarify this paper. This research was funded by The National Key Research and Development Program of China (2016YFB0201305, 2016YFB0200803, 2016YFB0200300), the National Natural Science Foundation of China under grant nos. (61521092, 91430218, 31327901, 61472395, 61432018). This research was also funded by the US Department of Energy, Office for Advanced Scientific Computing (ASCR) under Award No. 66150: "CENATE: The Center for Advanced Technology Evaluation". Pacific Northwest National Laboratory (PNNL) is a multiprogram national laboratory operated for DOE by Battelle Memorial Institute under Contract DE-AC05-76RL01830.

References

[1] 2017. *Network Repository*. http://networkrepository.com/networks.php
[2] 2018. *Comparison with Other Engines*. https://gunrock.github.io/docs/engines_topc.html
[3] September 2016. *nvgraph*. https://developer.nvidia.com/nvgraph
[4] September 2017. *graph500*. http://www.graph500.org/
[5] Jason Ansel, Cy P. Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman P. Amarasinghe. 2009. PetaBricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. 38–49. https://doi.org/10.1145/1542476.1542481

[6] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. 781–792.

[7] Scott Beamer, Krste Asanović, and David Patterson. 2013. Direction-optimizing breadth-first search. *Scientific Programming* 21, 3-4 (2013), 137–148.

[8] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *ACM Sigplan Symposium on Principles and Practice of Parallel Programming*. 235–248.

[9] Maciej Besta, MichaÅĆ Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *The International Symposium*. 93–104.

[10] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[11] Sergey Brin and Lawrence Page. 2012. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer networks* 56, 18 (2012), 3825–3833.

[12] Jee W Choi, Amik Singh, and Richard W Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM sigplan notices*, Vol. 45. ACM, 115–126.

[13] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *IEEE International Parallel and Distributed Processing Symposium*. 349–359.

[14] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May OâĂŹReilly, and Saman Amarasinghe. 2015. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 379–390.

[15] Jens Domke, Torsten Hoefler, and Wolfgang E Nagel. 2011. Deadlock-free oblivious routing for arbitrary topologies. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 616–627.

[16] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[17] Evangelos Georganas, Aydin Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2014. Parallel de bruijn graph construction and traversal for de novo genome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 437–448.

[18] Abdullah Gharaibeh, Lauro BeltrÃčo Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2017. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *International Conference on Parallel Architectures and Compilation Techniques*. 345–354.

[19] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *OSDI*, Vol. 12. 2.

[20] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–13.

[21] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. *Acm Sigplan Notices* 46, 8 (2011), 3–12.

[22] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P Sadayappan. 2017. MultiGraph: Efficient Graph Processing on GPUs. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE, 27–40.

[23] Sungpack Hong, Kyun Kim Sang, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. *Acm Sigplan Notices* 46, 8 (2011), 267–276.

[24] Rashid Kaleem, Anand Venkat, Sreepathi Pai, Mary Hall, and Keshav Pingali. 2016. Synchronization Trade-Offs in GPU Implementations of Graph Algorithms. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*. 514–523.

[25] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical foundations of the GraphBLAS. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 1–9.

[26] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2016. Scalable SIMD-Efficient Graph Processing on GPUs. In *International Conference on Parallel Architecture and Compilation*. 39–50.

[27] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha:vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 239–252.

[28] John Kloosterman, Jonathan Beaumont, Mick Wollman, Ankit Sethia, Ron Dreslinski, Trevor Mudge, and Scott Mahlke. 2015. WarpPool: sharing requests with inter-warp coalescing for throughput processors. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 433–444.

[29] Jérôme Kunegis and Julia Preusse. 2012. Fairness on the web: Alternatives to the power law. In *Proceedings of the 4th Annual ACM Web Science Conference*. ACM, 175–184.

[30] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a pc. USENIX.

[31] Da Li and Michela Becchi. 2013. Deploying Graph Algorithms on GPUs: An Adaptive Solution. In *IEEE International Symposium on Parallel & Distributed Processing*. 1013–1024.

[32] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 117–126.

[33] Hang Liu and H. Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *International Conference for High PERFORMANCE Computing, Networking, Storage and Analysis*. 68.

[34] Junhong Liu, Xin He, Weifeng Liu, and Guangming Tan. [n. d.]. Register-Aware Optimizations for Parallel Sparse Matrix-Matrix Multiplication. *International Journal of Parallel Programming* ([n. d.]).

[35] Junhong Liu, Xin He, Weifeng Liu, and Guangming Tan. 2018. Register-based Implementation of the Sparse General Matrix-matrix Multiplication on GPUs. *SIGPLAN Not.* 53, 1 (Feb. 2018), 407–408. https://doi.org/10.1145/3200691.3178529

[36] Yixun Liu, Eddy Z Zhang, and Xipeng Shen. 2009. A cross-input adaptive framework for GPU program optimizations. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 1–10.

[37] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E. Guestrin, and Joseph Hellerstein. 2014. GraphLab: A New Framework For Parallel Machine Learning. *Computer Science* (2014).

[38] Yulong Luo, Guangming Tan, Zeyao Mo, and Ninghui Sun. 2015. FAST: A fast stencil autotuning framework based on an optimal-solution space model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 187–196.

[39] Grzegorz Malewicz, Matthew H. Austern, Aart J. C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data*. 135–146.

[40] Adam Mclaughlin and David A. Bader. 2015. Scalable and High Performance Betweenness Centrality on the GPU. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. 572–583.

[41] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *Acm Sigplan Notices* 47, 8 (2012), 117–128.

[42] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[43] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Atomic-free irregular computations on GPUs. In *The Workshop on General Purpose Processor Using Graphics Processing Units*. 96–107.

[44] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-driven versus topology-driven irregular computations on GPUs. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 463–474.

[45] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D Owens. 2017. Multi-GPU graph analytics. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 479–490.

[46] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 472–488.

[47] Ahmet Erdem SariyÃijce, Kamer Kaya, Erik Saule, and ÃIJmit V ÃĞatalyÃijrek. 2013. Betweenness centrality on GPUs and heterogeneous architectures. *Advances in Journalism & Communication* 01, 4 (2013), 50–53.

[48] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P Sadayappan. 2015. Automatic selection of sparse matrix representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 99–108.

[49] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: processing large-scale graphs on accelerator-based systems. (2015), 1–12.

[50] Xuanhua Shi, Junling Liang, Sheng Di, Bingsheng He, Hai Jin, Lu Lu, Zhixiang Wang, Xuan Luo, and Jianlong Zhong. 2015. Optimization of asynchronous graph processing on GPU with hybrid coloring model. In *Acm Sigplan Symposium on Principles & Practice of Parallel Programming*. 271–272.

[51] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: a survey. *ACM Computing Surveys (CSUR)* 50, 6 (2018), 81.

[52] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *Acm Sigplan Notices* 48, 8 (2013), 135–146.

[53] Jyothish Soman, Kothapalli Kishore, and P J Narayanan. 2010. A fast GPU algorithm for graph connectivity. (2010), 1–8.

[54] Nikita Spirin and Jiawei Han. 2012. Survey on web spam detection: principles and algorithms. *ACM SIGKDD Explorations Newsletter* 13, 2 (2012), 50–64.

[55] Bryan Thompson, Bryan Thompson, and Bryan Thompson. 2014. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *The Workshop on Graph Data Management Experiences and Systems*. 1–6.

[56] Merijn Verstraaten, Ana Lucia Varbanescu, and Cees de Laat. 2017. Using Graph Properties to Speed-up GPU-based Graph Traversal: A Model-driven Approach. *arXiv preprint arXiv:1708.01159* (2017).

[57] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 389–404.

[58] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2015. Gunrock: a high-performance graph processing library on the GPU. In *Acm Sigplan Symposium on Principles & Practice of Parallel Programming*. 265–266.

[59] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. 2017. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing (TOPC)* 4, 1 (2017), 3.

[60] R Clint Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 1–27.

[61] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In *ACM Sigplan Symposium on Principles and Practice of Parallel Programming*. 194–204.

[62] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, et al. 2017. Big graph analytics platforms. *Foundations and Trends® in Databases* 7, 1-2 (2017), 1–195.

[63] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 31–43. https://doi.org/10.1145/3018743.3018755

[64] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. [n. d.]. Bridging the gap between deep learning and sparse matrix format selection.

[65] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel & Distributed Systems* 25, 6 (2014), 1543–1552.

[66] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System.. In *OSDI*. 301–316.