Mix-and-Match: A Model-driven Runtime Optimisation Strategy for BFS on GPUs

Merijn Verstraaten*, Ana Lucia Varbanescu† & Cees de Laat‡
Informatics Institute, University of Amsterdam
Amsterdam, The Netherlands
{m.e.verstraaten, a.l.varbanescu, delaat}@uva.nl

Abstract—It is universally accepted that the performance of graph algorithms is heavily dependent on the algorithm, the execution platform, and the structure of the input graph. This variability remains difficult to predict and hinders the choice of the right algorithm for a given problem.

In this work, we focus on a case study: breadth-first search (BFS), a level-based graph traversal algorithm, running on GPUs. We first demonstrate the severity of this variability by presenting 32 variations of 5 implementation strategies for GPU-enabled BFS, and showing how selecting one single algorithm for the entire traversal can significantly limit performance. To alleviate these performance losses, we propose to mix-and-match, at runtime, different algorithms to compose the best performing BFS traversal. Our approach is based on two novel elements: a predictive model, based on a decision tree, which is able to dynamically select the best performing algorithm for each BFS level, and a quick context switch between algorithms, which limits the overhead of the combined BFS.

We demonstrate empirically that our dynamic switching BFS outperforms our non-switching implementations by 2× and existing state-of-the-art GPU BFS implementations by 3×. We conclude that mix-and-match BFS is a competitive approach for performing fast graph traversal, while being easily extended to include more BFS implementations and easily adaptable to other types of processors or specific types of graphs.

I. Introduction

Graph processing is an important part of data science, due to the flexibility of graphs as models for highly interrelated data. Given the rapid growth of dataset sizes, as well as the expected complexity increase of graph processing applications, a lot of research focuses on parallel and distributed solutions for graph processing [1, 2, 3, 4]. Fuelled by the high-performance potential of graphics processing units (GPUs), novel ways to circumvent graph processing challenges [5] to accommodate the massive parallelism of GPUs have also emerged [6, 7, 8, 9, 10].

The performance of graph processing is affected by both the underlying hardware and the structure of the input graph. While it is accepted as common knowledge that the performance of data-dependent algorithms is impacted by the structure of the data, little progress has been made in understanding how large this impact actually is. Similarly, the correlation between structural properties of the input graph and the observed performance remains poorly understood [11, 12, 13]. Comprehensive workload characterization for graph processing on parallel systems has been attempted [14], but not for GPUs. Finally, no conclusive work exists on analytical modelling for parallel graph processing, either.

In this work we take a data-driven approach to analysing the impact structural graph properties have on graph processing performance. To do so, we collect performance data from a wide set of real world graphs from the KONECT repository [15], and use them as training data for a binary decision tree (BDT) model. For this problem, BDTs offer a good balance between accuracy and prediction speed, thus enabling runtime switching of algorithms for each traversal level, ultimately enabling performance gain for our mix-and-match BFS. While this approach provides little explicit insight into the actual correlations between graph properties and BFS performance, it does provide a systematic process for building a prediction model, and many tuning possibilities in terms of features, variables, and actual methods.

We have applied this model on the 246 graphs from KONECT; for each graph we performed traversals starting from 11 different starting vertices, and collected the features and performance indicators for each traversed level. We train the model on a uniform random selection of 60% of the data points. We use the model as a switch predictor for a level-switching adaptive BFS. With this adaptive BFS, we outperform two popular graph processing systems for GPUs: we gain on average 4.5× over Gunrock [7] and 45× over LonestarGPU [16].

The main contributions of this paper are the following:

- We show that the performance of different BFS implementations varies dramatically not only across graphs, but also during the BFS traversal of one graph, with differences of up to two orders of magnitude (Section III).
- We create a binary decision tree model that can predict which implementation to use at every BFS level. (Section IV-A).
- We demonstrate that our decision tree model is accurate enough and fast enough to evaluate online, allowing for dynamic runtime switching of implementations (Section IV-C).
- We demonstrate that we can use our prediction model to perform dynamic switching, thus obtaining a mix-and-match BFS able to perform, on average, 2× faster than the best single-implementation BFS

(Section IV-C).

II. Background

For readers unfamiliar with BDTs or GPGPU processing, this section provides basic information required to understand the rest of this paper.

A. Graph Processing

Graphs are collections of entities (called nodes or vertices) and relationships between them (called edges) — $G = (V, E)$. Graph processing typically implies some transformation of the original graph by traversing its edges and visiting its nodes.

The simplest example is a traversal itself, where, given a starting node (i.e., the root node), the algorithm has to visit all nodes accessible from the root, eventually saving the shortest path between the root and each accessible node. Typically, two types of traversals exist: Breadth-First Search (BFS) and Depth-First Search (DFS). In this work we focus on BFS, which is a level-based traversal: the graph is discovered level-by-level, with all the neighbours of the current frontier (the most recently discovered level) forming the next level.

In general, graph processing applications, like BFS, are difficult to parallelize due to their properties: low compute-to-communication ratio, data-dependent behaviour, poor data locality, variable parallelism, and load imbalance [5]. Thus, proposing efficient solutions for parallel graph processing algorithms remains a challenge. In this work, we focus on parallel graph processing using GPUs.

B. General Purpose GPU Programming

GPUs are massively parallel architectures providing fine-grained data parallelism. Our work focuses on understanding the potential for GPUs to boost the performance of graph processing algorithms, which are notoriously difficult to parallelize efficiently. In this work we use NVIDIA GPUs, due to the superior programmability provided by CUDA, the native NVIDIA GPU programming model. We note however that the proposed method is independent of the programming model and parallel platform[1].

The idea behind the programming model is simple: CUDA provides a mapping of the programming model concepts onto the hardware, while preserving a sequential programming model per thread.

For the actual computation, programmers focus on implementing the single-threaded code, called a kernel; they further write the host code to launch enough threads to (1) cover the space of the problem, and (2) provide enough potential for latency hiding [17].

The threads that execute the kernel are grouped into thread blocks, which are scheduled on the streaming multiprocessors. All blocks form a grid, containing all the logical threads that are to be scheduled and executed on the cores themselves.

In terms of the execution model, NVIDIA GPUs work with warps. A warp is a group of 32 threads that work in lock-step: they all execute the same instruction on multiple data. This model is called Single-Instruction, Multiple Thread (SIMT) and enables high performance by massive parallelism. However, it is unable to handle diverging threads; as a result, load-imbalance between threads within the same warp introduces severe performance penalties. Aside from thread divergence, there are also performance challenges related to the use of (global) atomic operations and lack of coalescing for main memory accesses.

Our software stack — the BFS kernels, required boilerplate code, and analysis tools — is based on C++ and CUDA, and it is available at GitHub[2].

C. Decision Trees

Decision trees are a non-parametric, supervised learning technique [18]. They come in two flavours, classifiers and regressors. We chose to use Binary Decision Trees to build our predictive model, because they are easy to embed into existing code, fast to evaluate, and can help inform our analytical modelling. For building the model we use the implementation in the Python library scikit-learn [19], which uses an algorithm based on CART [18]. We then generate a C++ implementation of the constructed BDT, which is used to perform the predictions at runtime.

Due to the way trees are constructed, overfitting issues can become more pronounced if the input parameters in the learning set are not uniformly distributed across the range we intend to predict against. Additionally, as the number of input parameters increases, it becomes exponentially more costly to compute the best discriminator, which in turn makes the algorithm slower and increases the risk of bias and overfitting. To avoid this problem we take the standard precaution of separating our dataset into a separate training and validation set, cross-validating our model against the unseen data points in the validation set. A detailed discussion on overfitting for our specific models is presented in Section IV-D.

III. Experiments

This section presents the bulk of our experimental results. We benchmarked our 32 different BFS implementations on the graphs from the KONECT [15] repository, measuring both the total time and the time taken for each level of BFS. We used these results to train and validate our Binary Decision Tree (BDT) model. We further used the model to dynamically switch between different variants at runtime, thus creating our high-performance, mix-and-match BFS.

[1]CUDA is the native programming model for NVIDIA GPUs; it is proprietary to NVIDIA, but has a huge ecosystem of libraries and helpful tools, unmatched by models like OpenCL or OpenACC.

[2]https://github.com/merijn/gpu-benchmarks

A. BFS Implementations

We implemented 5 different neighbour-iteration primitives, and created several variants for each of these primitives. These 5 neighbour-iteration primitives consist of: 2 edge-centric implementations (edge list and reverse edge list), 2 vertex-centric implementations (vertex push and vertex pull), and 1 virtual warp-based implementation inspired by the work of Hong, et al. [20]. The different implementations for each primitive differ in the way the new frontier is computed at the end of each BFS level, or the virtual warp configuration. In this subsection we describe how these versions differ from each other.

Each algorithm starts by initialising all depths to infinity, then initialising the root node's depth to 0. During every level of BFS we compute the frontier size, that is, the number of vertices that have been assigned a new depth.

1) Edge List & Reverse Edge List: For every level of BFS these edge-centric implementations launch one CUDA thread per edge. If the depth of the origin vertex is equal to the current BFS level, then the depth of the destination vertex is updated to the minimum of its current depth and the BFS level plus one.

The edge list implementation uses the outgoing edges of every vertex, whereas the reverse edge list implementation use the incoming edges of every vertex. This difference affects the amount of memory coalescing and the access patterns exhibited at runtime.

The advantage of these edge-centric parallelisations is that they never suffer from workload imbalance, every thread in a warp performs the same amount of work. The fact that many threads have to read the depth of the same origin vertex helps with coalescing memory access. The downside is that they result in many parallel updates, resulting in many contested atomic updates.

2) Vertex Push & Vertex Pull: For every level of BFS, our vertex-centric implementations launch one CUDA thread per vertex. For the push implementation, if the vertex's depth is equal to the current BFS level, the thread iterates over all its neighbours, updating their depth to the minimum of their current depths and the BFS level plus one. For the pull implementation, if the vertex has no depth yet, the thread iterates over its neighbours until it encounters one whose depth matches the current BFS level; if this happens, the depth of the original node is set to the current BFS level plus one.

Both implementations are susceptible to workload imbalance, and thus performance loss, if vertices with wildly different degrees are in the same warp. The push version, similar to edge-centric ones, generates a lot of concurrent updates, requiring a many atomic operations. However, if the frontier is small, it avoids many useless reads, since the depth of every vertex is only read once.

The pull version requires no atomics as the depth of a vertex is only ever touched by one thread. But if none of the neighbours of a vertex are in the frontier, a lot of time is wasted iterating over neighbours for nothing. As such, vertex-pull is, intuitively, more efficient for a large frontier, as the likelihood of a neighbour being in the frontier scales with frontier size.

3) Vertex Push Warp: Rather than assigning one thread per vertex, this method uses the virtual warp approach described in [20], which attempts to mitigate the negative impact produced by workload imbalance between threads.

The basic principle is the same as with vertex push, but we divide the warps into smaller "virtual warps". Each virtual warp gets a number of vertices equal to its number of threads. However, instead of each thread processing the edges for one vertex, all threads process the edges of a single vertex in parallel. This repeats until all vertices assigned to the virtual warped have been processed.

This reduces the amount of load imbalance occurring within a virtual warp, since the workload of a virtual warp is spread out equally across that virtual warp. However, the optimal size of the virtual warp is challenging to determine. Moreover, the different graphs and even different levels of the graph also require tuning of the warp sizes for best performance.

4) Variants: In every BFS level, zero or more new vertices get discovered, forming the frontier for the next level. We need to track the size of the frontier, since the algorithm terminates when no new nodes are discovered. We do this by enabling each thread to track how many new vertices it has discovered, and aggregating these counts at the end of each BFS level to compute the new frontier size.

We implemented four different aggregation methods. The first variant uses a global counter and every thread performs an atomic addition on this counter. The second variant tries to alleviate the atomic operation penalty by batching the atomics performed by a single thread. The literature suggests that the number of atomic operations and contention can be reduced further by performing a reduction within a warp or block [21] before performing the global atomic operation. Thus, the third and fourth variants perform a warp and a warp-and-block reduction, respectively, before atomically updating the frontier size.

B. Experimental Setup

All measurements were done on an NVIDIA TitanX, using version 8.0 of the CUDA toolkit. The source code of these benchmarks can be found on GitHub[3].

As for datasets, we retrieved all the graphs from the KONECT repository and ran each of the 32 implementations described above on all of them. Additionally, for every graph, we used 11 different root vertices. All results presented here are averaged over 30 runs, and exclude input data and result transfer times. For readability reasons, we only show 1 variant for each neighbour iteration primitive, as otherwise the graphs become too cluttered to read.

Figure 1 shows the runtimes, normalised to the slowest implementation for each graph, for a selection of KONECT graphs.

[3]https://github.com/merijn/GPU-benchmarks

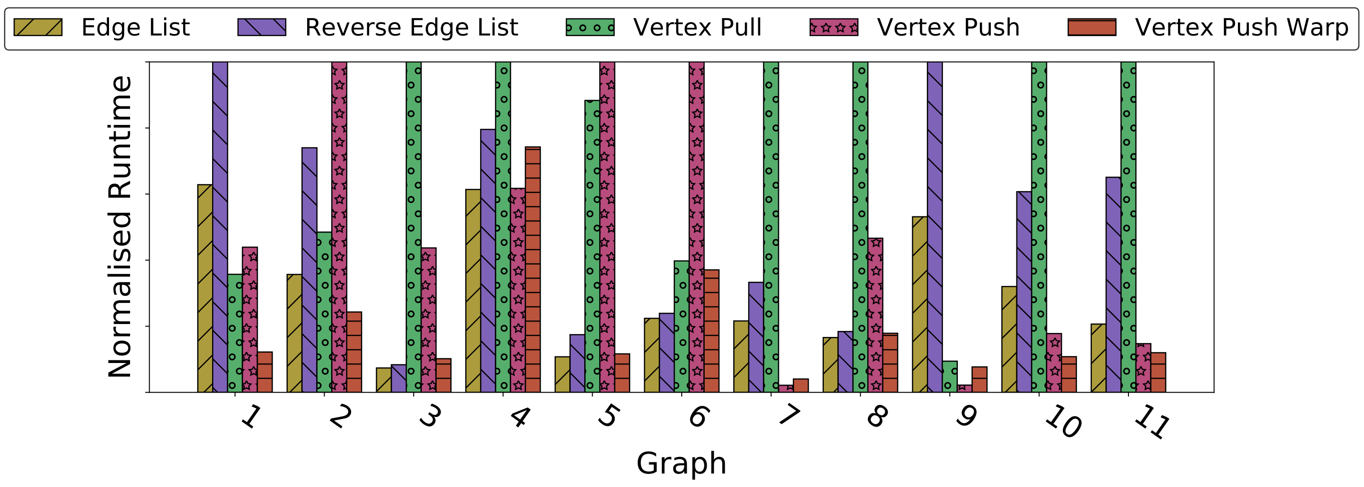

Fig. 1: Total runtimes, normalised to the slowest time, for different BFS implementations for a selection of 11 KONECT graphs. See Table I for input graphs details.

No.	Graph	# Vertices	# Edges
1	actor-collaboration	382,219	30,076,166
2	ca-cit-HepPh	28,093	6,296,894
3	cfinder-google	15,763	171,206
4	dbpedia-starring	157,183	562,792
5	discogs_affiliation	2,025,594	10,604,552
6	opsahl-ucsocial	1,899	20,296
7	prosper-loans	89,269	3,330,225
8	web-NotreDame	325,729	1,497,134
9	wikipedia_link_en	12,150,976	378,142,420
10	wikipedia_link_fr	3,023,165	102,382,410
11	zhishi-hudong-internallink	1,984,484	14,869,484

TABLE I: Details for the input graphs shown in Figure 1 and Figure 3.

Figure 1 clearly illustrates the performance variability of different implementations: performance can vary by orders of magnitude across input graphs.This effectively means that when (accidentally) choosing the worst algorithm, one can loose 1–2 orders of magnitude in performance for a BFS traversal compared with the best option. Thus, an informed decision about the algorithm to be used for a given graph very important for any efficiency metric. However, this is no easy task: no models are available to determine the best or the worst algorithm for a given graph traversal task.

One of the reasons for which predicting the best algorithm for the entire graph is difficult is the huge performance difference that can be observed during traversing a single graph: (1) between two different BFS levels in the same graph, the performance of the same algorithm can vary up to an order of magnitude, and (2) per-level, the differences between different algorithms can be up to four orders of magnitude. These large performance gaps, are illustrated in fig. 2, which presents an example of the performance of the five main BFS implementations, per-level, for the actor-collaborations graph. We see that the vertex push and pull are best for most levels, but on a couple of levels they perform so badly that their overall performance becomes below par. Such behaviour is a strong indication that switching algorithms at every level might be even better than devising a model to detect the best overall solution.

IV. Modeling BFS Performance

In this section we describe the training processes used by our automated, parallel toolchain to build our decision tree model, we discuss its accuracy and applicability for online performance prediction, and evaluate the feasibility of a dynamically switching BFS.

A. Building the model

Before proceeding to build our model, we need to decide on the features we will use and provide a suitable dataset for training.

There is no consensus on which of a graph's structural properties impact the performance of graph algorithms. Our previous modelling attempts in [12], combined with our experience while optimising and developing our implementations, lead us to believe that the graph size and degree distribution are the biggest impact factors when it comes to BFS traversal.

Additionally, work on adaptive BFS [3, 10] and the observed runtime changes across levels, indicate that behaviour at each level depends on the size of the frontier discovered in the previous level, and the percentage of the graph that has already been discovered.

Therefore, we consider the following relevant features for our model:

Graph size: the number of vertices and edges in the graph.

Frontier size: either as absolute number of vertices or as percentage of the graph's vertices.

Discovered vertex count: either as absolute number of vertices or as percentage of the graph's vertices.

Degree distribution: represented by the 5 number summary and standard deviation of in, out, or absolute degrees.

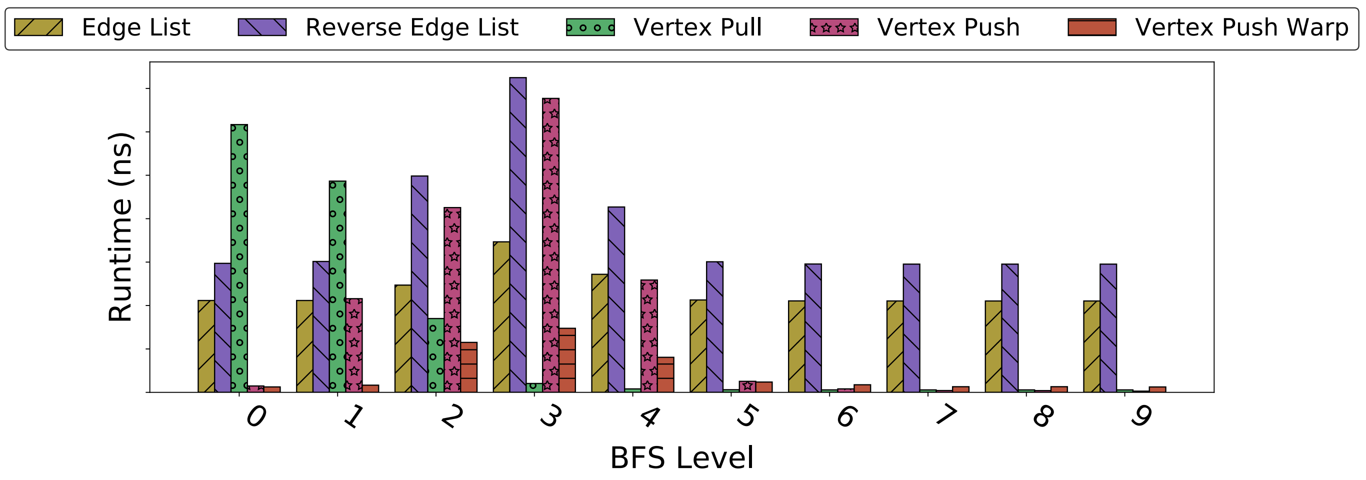

Fig. 2: Runtimes of different BFS implementations per level of the actor-collaborations graph from KONECT.

Our experiments (see Section III) provided us with performance data for all of our 32 implementations and each BFS level of all KONECT graphs. These data allow us to determine the fastest implementation for each BFS level of each graph. From this data we built a dataset where we associate every measured BFS level with the structural properties of the graph it was run on, and the level specific information. Of this dataset we used 60% as a training set, keeping the remaining 40% as a validation set.

The models described in the rest of this section consist of binary decision trees trained to predict the best performing algorithm for a given level of BFS, based on a mix of the above properties.

B. Model Accuracy

We define the optimal BFS traversal of a graph as the traversal where the fastest of our implementations is used at every level. To evaluate the accuracy of our model, we take this optimal runtime as a reference (i.e., as 1) and evaluate the predicted and observed runtimes as a slowdown compared to this reference. The larger the gap, the further away we are from the optimal performance.

In table II we compare the model's predictions and the different implementations against the optimal runtime across all KONECT graphs. The optimal runtime is the execution time of the optimal BFS traversal. The "non-switching best" runtime shows the numbers if an oracle lets us pick the fastest non-switching algorithm for each graph ahead of time.

The mix-and-match implementation, based on our model's predictions, leads to an average runtime of 2.04 of optimal — effectively, a 100% slowdown compared to optimal. The average runtime for always picking the best non-switching implementation is 2.44, or 144% slowdown compared to optimal. In other words, our mix-and-match BFS can obtain a 40% speed-up compared to the fastest non-switching implementation. In practice, the potential gain is even more significant, because no model or oracle exists to selecting the fastest non-switching implementation.

Algorithm	1–2×	>5×	>20×	Average	Worst
Mix-and-Match	92%	2.5%	0.4%	2.04×	498×
Non-switching Best	65%	8%	0%	2.44×	37×
Edge List	49%	22%	2.2%	4.16×	61×
Rev. Edge List	39%	33%	8.8%	7.04×	108×
Vertex Pull	16%	58%	30%	48.41×	2,495×
Vertex Push	23%	53%	28%	55.61×	1,980×
Vertex Push Warp	18%	25%	4.9%	5.42×	88×

TABLE II: Algorithm performance compared to theoretical optimum over all the graphs in KONECT.

C. Empirical Results

To verify whether the predictions described above are actually achievable, we implemented "Mix-and-Match", an adaptive BFS implementation able to switch between implementations on-the-fly, based on our model's predictions.

Most of our implementations operate on different representations of the graph. Thus, switching between implementations also involves switching between in-memory representations. To do so, we need to either (a) generate/bring the new representation in memory on-the-fly, or (b) keep all representations in memory.

We considered option (a) infeasible, as transferring data to and from the GPU is generally slow, and doing so for each level would become prohibitive, performance-wise. Instead, we chose to consider this a classical time-space trade-off, where we trade memory for faster computation and keep each necessary representation of the graph in memory.

The two main graph representations we use are a Compressed Sparse Row (CSR) for the vertex-centric implementations, and an edge list for the edge-centric implementations. We can combine the two by simply storing the origin vertex for every edge in our CSR. This increases the storage from 1 int/vertex and 1 int/edge (for

CSR) and 2 int/edge (for edge list), to 1 int/vertex and 2 int/edge. This is not very expensive, memory-wise: it is a mere 38 MiB for a graph of 10,000,000 edges.

Our adaptive Mix-and-Match implementation achieved runtimes that were within 5% of our prediction across the entire dataset, resulting in numbers comparable to the predictions listed in table II.

These results show that our model leads to considerable speed-up compared to our individual implementations. However, speed-up results are only as good as the baseline. Thus, we further compare the mix-and-match results against two existing GPU graph processing frameworks to establish how much "real world" performance we gain by using the model described in this paper.

Figure 3 compares our results against the state-of-the-art GPU graph processing framework Gunrock [7] and the slightly older BFS benchmark LonestarGPU [16], across a selection of KONECT graphs. We benchmarked both Gunrock and LonestarGPU on the same hardware, using 148 different KONECT graphs. On average, Gunrock achieves a performance of 6.5× of our theoretical optimum. LonestarGPU manages 63× of optimal. Our model's 2× of optimal means that we are, on average, 3× faster than Gunrock.

D. Overfitting & Generality Concerns

As mentioned in section II, we took the standard precaution of training our model against a subset of 60% of our data and validating its accuracy against a separate test set of 40% of the data points. In this validation, the model accurately predicts the fastest algorithm in 70% of the cases. Moreover, in the majority of the "mispredicted" 30%, the model is often predicting an implementation with similar performance to the correct prediction.

From this data we can conclude that the model's accuracy is high with regards to our KONECT repository of graphs. However, we expect the portability of the model to be highly correlated with how representative the training set is for the test set. For example, if we train the model on social networks graphs only, we expect it to be better at predicting the best BFS for social networks, and less so for, say, road networks. Therefore, we recommend that the actual training and modelling process is driven by the prediction goals.

For example, if the goal is to build a generic model to predict most graphs, using a large variety of graphs for training is mandatory. A collection like KONECT is a good start, but while validating the model against the KONECT data set, we noticed that bad model predictions are correlated with several classes of graphs, such as bipartite graphs, and graphs with extremely skewed degree distributions, which are less represented in the repository (and, thus, in our training data).

On the other hand, if the goal is to have a model tweaked for a specific type of graphs — e.g., social or road networks — only a subset of the graphs in public repositories can be useful for training. Whether there are sufficient such graphs depends on many factors. However, this analysis deserves a dedicated study, focused on determining what is the ideal size and composition of a specialized training set; such a study is beyond the scope of this work.

To summarize, we make no portability claims or guarantees of the trained model for more specialized repositories, and we recognize the limitations of our training dataset. However, the training and prediction processes are systematic, straightforward, and generic, and can be easily applied again for different training data/environments, eventually improving/tuning the predictor to match the goal.

V. Related Work

We summarize in this section related work on parallel BFS implementations and on the use of machine learning models for performance prediction, which have both inspired this work.

A. Algorithms

Despite the advances in large-scale graph traversal algorithms, like direction switching BFS [3], distributed-memory BFS [4], and the matrix-based graph processing solution [22], there's still no single best BFS traversal solution. This happens because BFS is highly dependent on the graph properties, with different algorithms and/or implementations eventually suffering from different bottlenecks. When combining this with complex, massive parallel machines like the GPUs [23], the performance gaps are even more difficult to predict.

In our work, we steer from devising a best BFS algorithm. Instead, we focus on selecting the best performing solution (from a given set) for each iteration. This is similar to [10], but our solution combines more algorithms and uses a more deterministic, systematic switching criterion. Moreover, our approach can be extended to incorporate additional BFS versions, as long as sufficient performance data are available for training.

B. Graph Processing Systems

The new challenges of graph processing have also reflected in the amount of systems and frameworks designed for efficient, high performance graph processing [24, 25]. From these systems, a handful of GPU-enabled systems have also emerged [7, 16], combining clever BFS algorithms with specific GPU-based optimisations [20]. Still, none of them can claim the absolute best performance for the same reason: the diversity of graphs and their properties lead to high performance variability for all these systems [26].

Our work is complementary to the above graph processing systems/frameworks: our switching approach can be, in principle, incorporated in any of these frameworks. Performance-wise, we are competitive against these systems (see Section IV-B).

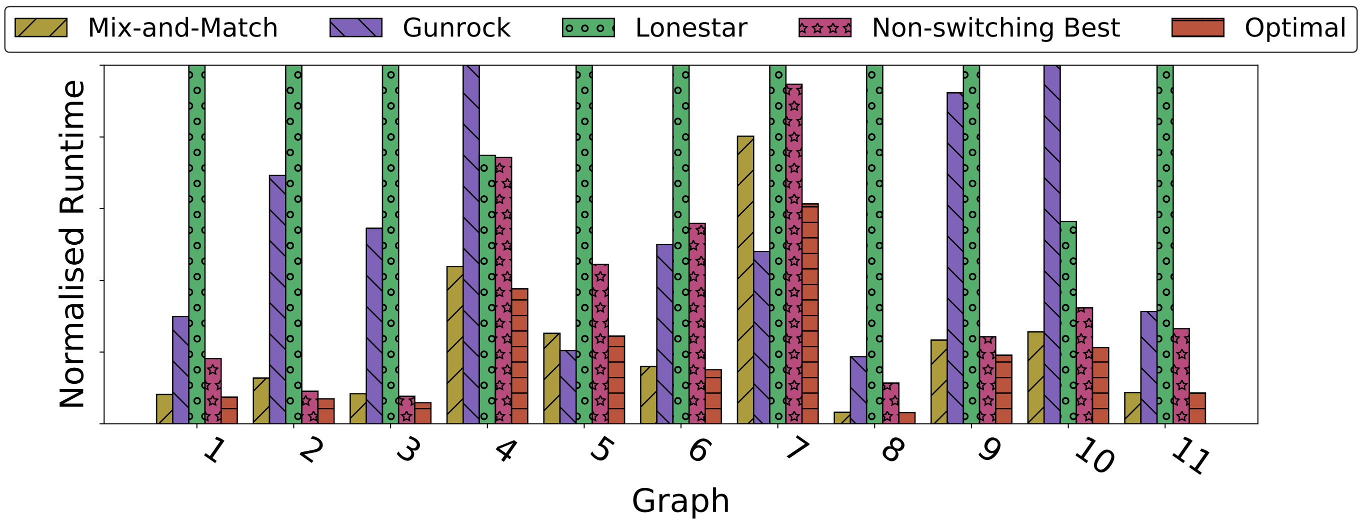

Fig. 3: Comparison of runtimes of different BFS implementations, predicted performance, and existing optimised BFS implementations on KONECT graphs. Normalised to the slowest runtime for each graph. See Table I for the details of the input graphs.

C. Machine Learning for Performance Modelling

Our mix-and-match BFS relies heavily on performance prediction, which in turn is based on a model built using machine learning. Performance prediction based on machine learning models has been successful in the past [27, 28, 29, 30, 31, 32], but applying machine learning for an adaptive, level-switching BFS requires significant changes. Specifically, an adaptive implementation requires that features and predictions are fast enough to compute to not lose any performance gain to this new overhead. To the best of our knowledge, we are the first to have trained and used such a model for improving BFS performance by runtime switching.

VI. Conclusion

The increased availability of large graphs and the high demand for their analysis have made GPUs a successful platform for graph processing. However, the performance GPUs can deliver for processing graphs is highly variable, depending on the input graph and chosen algorithm. So far, this variability has been difficult to quantify.

In this work, we propose to quantify and further use this variability to gain performance for a BFS traversal. Our approach works as follows: given a set of BFS algorithms (32 in this work), and an input dataset, we predict and employ, for each level in the BFS traversal, the best algorithm in the available set. This is a generalization of the work on direction-switching BFS [3] and adaptive graph algorithms [10], to which we have added a much more systematic switching detection.

Our switching strategy is based on a decision tree model, which is trained to predict, at runtime, which BFS variant is the best for the next iteration. This combination of machine learning modelling and the large set of algorithms we use makes our approach competitive with state-of-the-art graph processing systems and algorithms.

Our results demonstrate our mix-and-match approach delivers high performance, with an average gain of 3× over Gunrock and 30× over LonestarGPU. Our mix-and-match is within $1--2\times$ of the absolute theoretical optimum in ∼92% of cases, and outperforms the (fictive) oracle that selects the best non-switching implementation ahead of time by ∼40%.

We conclude that this work is a step forward in quantifying and using the impact of graph properties on the performance of graph processing. Our future work will investigate the potential contribution of adding more BFS algorithms, and test other modelling techniques that offer a good balance between accuracy and speed of runtime evaluation. Finally, on the long term, we plan to expand this approach to other parallel platforms and graph processing algorithms.

References

[1] M. Capotă, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz, “Graphalytics: A big data benchmark for graph-processing platforms,” in Proceedings of the GRADES'15, ser. GRADES'15. ACM, 2015, pp. 7:1–7:6.

[2] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, “How Well do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis,” in IPDPS, 2014.

[3] S. Beamer, K. Asanović, and D. Patterson, “Direction-optimizing breadth-first search,” Scientific Programming, vol. 21, no. 3-4, pp. 137–148, 2013.

[4] A. Buluç, S. Beamer, K. Madduri, K. Asanović, and D. Patterson, “Distributed-memory breadth-first search on massive graphs,” in Parallel Graph Algorithms, D. Bader, Ed. CRC Press, Taylor-Francis, 2016 (in press). [Online]. Available: http://gauss.cs.ucsb.edu/~aydin/ChapterBFS2015.pdf

[5] B. H. A. Lumsdaine, D. Gregor and J. W. Berry, "Challenges in parallel graph processing," Parallel Processing Letters 17, 2007.

[6] Y. Guo, A. L. Varbanescu, A. Iosup, and D. Epema, "An Empirical Performance Evaluation of GPU-Enabled Graph-Processing Systems," in CCGrid'15, 2015.

[7] 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 Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2016, p. 11.

[8] S. Heldens, A. L. Varbanescu, and A. Iosup, "Dynamic load balancing for high-performance graph processing on hybrid cpu-gpu platforms," in IA3 2016 (SC-Workshops), Nov 2016.

[9] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: vertex-centric graph processing on GPUs," in HPCS. ACM, 2014, pp. 239–252.

[10] D. Li and M. Becchi, "Deploying graph algorithms on gpus: An adaptive solution," in IPDPS 2013, May 2013, pp. 1013–1024.

[11] A. L. Varbanescu, M. Verstraaten, A. Penders, H. Sips, and C. de Laat, "Can Portability Improve Performance? An Empirical Study of Parallel Graph Analytics," in ICPE'15, 2015.

[12] M. Verstraaten, A. L. Varbanescu, and C. de Laat, "Quantifying the performance impact of graph structure on neighbour iteration strategies for pagerank," in Euro-Par 2015: Parallel Processing Workshops. Springer, 2015, pp. 528–540.

[13] C. Lehnert, R. Berrendorf, J. P. Ecker, and F. Mannuss, "Performance prediction and ranking of spmv kernels on gpu architectures," in European Conference on Parallel Processing. Springer, 2016, pp. 90–102.

[14] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in IISWC 2015, Oct 2015.

[15] J. Kunegis, "Konect: The koblenz network collection," in Proceedings of the 22Nd International Conference on World Wide Web, ser. WWW '13 Companion, 2013, pp. 1343–1350.

[16] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in Workload Characterization (IISWC), 2012 IEEE International Symposium on. IEEE, 2012, pp. 141–151.

[17] V. Volkov, "Understanding latency hiding on gpus," 2016.

[18] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, Classification and Regression Trees. CRC press, 1984.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[20] T. O. S. Hong, S. K. Kim and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in PPoPP'11, 2011.

[21] M. Harris, "High Performance Computing With CUDA," 2007.

[22] A. Buluç, J. R. Gilbert, and V. B. Shah, "Implementing Sparse Matrices for Graph Algorithms," in Graph Algorithms in the Language of Linear Algebra, J. Kepner and J. R. Gilbert, Eds. SIAM Press, 2011.

[23] A. Buluç, J. R. Gilbert, and C. Budak, "Solving path problems on the GPU," Parallel Computing, vol. 36, no. 5-6, pp. 241 – 253, 2010. [Online]. Available: http://gauss.cs.ucsb.edu/publication/parco_apsp.pdf

[24] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," ACM Comput. Surv., vol. 48, no. 2, Oct 2015.

[25] N. Doekemeijer and A. L. Varbanescu, "A survey of parallel graph processing frameworks„" TUDelft, Tech. Rep. PDS-2014-003, 2014. [Online]. Available: http://www.ds.ewi.tudelft.nl/fileadmin/pds/reports/2014/PDS-2014-003.pdf

[26] Y. Guo, A. L. Varbanescu, A. Iosup, and D. Epema, "An empirical performance evaluation of gpu-enabled graph-processing systems," in Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on. IEEE, 2015, pp. 423–432.

[27] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ATI GPU: A statistical approach," in NAS'11. IEEE Computer Society, 2011.

[28] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent GPU architectures," in IPDPS '13. IEEE Computer Society, 2013.

[29] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in PPoPP'07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229479

[30] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis," in International Symposium on High-Performance Computer Architecture, 2006, pp. 99–108.

[31] G. Wu, J. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on, Feb 2015, pp. 564–576.

[32] S. Madougou, A. L. Varbanescu, C. D. Laat, and R. V. Nieuwpoort, "A tool for bottleneck analysis and performance prediction for gpu-accelerated applications," in 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2016, pp. 641–652.