Speeding Up GPU Graph Processing Using Structural Graph Properties

Merijn Verstraaten
University of Amsterdam
m.e.verstraaten@uva.nl

ACM Reference format:
Merijn Verstraaten. 2017. Speeding Up GPU Graph Processing Using Structural Graph Properties. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 3 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROBLEM STATEMENT

Due to its flexibility and wide applicability, graph processing is an important part of data science. With the prevalence of "big data", scaling increasingly complex analytics to increasingly large datasets is one of the fundamental problems in graph processing.

At the same time, hardware platforms are becoming increasingly parallel and heterogeneous. Distributed systems and accelerator-based architectures (e.g., based on Graphical Processing Units — GPUs, or Xeon Phi) are frequently cited as solutions for handling large compute workloads, even for graph processing [1, 11].

Both partitioning the data and efficient execution of graph operations on parallel and distributed systems remain hard problems. The heterogeneity of the available platforms makes matters worse, because different types of platforms require different approaches to perform in their "comfort zone".

I focus on the performance of graph operations on GPUs, seen as representative massively parallel HPC architectures. In this context, I start from the following observations:

(1) Speeding up graph processing by using GPUs requires efficient exploitation of the fine-grained parallelism of graph problems [6, 12].
(2) The efficiency in using the massive hardware parallelism (hundreds of cores) is highly dependent on the data locality and the regularity of both operations and data access patterns [14, 15].
(3) The data locality, the regularity of operations and data access patterns are highly dependent on *both* the in-memory representation of the data and the structure of the underlying graph.
(4) Most high-level graph operations support different implementations, with different memory representation and access patterns [4].

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In summary, given a high-level graph processing operation, there are multiple ways to implement it. Which implementation is the most efficient on a given platform is highly dependent on the structure of the graph being processed [15]. While this is common knowledge, there has been little work on how to predict the right implementation for a given input. Selecting the right implementation for the workload and hardware at hand is important because the performance gap between implementations can be of orders of magnitude.

The focus of my research has been on the following topics: *1*) Modelling the sequential workload of graph algorithm implementations and finding a parallel execution model for them, *2*) identifying which *structural properties*[1] of graphs impact implementation performance and quantifying this impact, *3*) identifying heuristics for algorithm implementation selection, *4*) determining the feasibility of dynamically switching between implementations during a computation.

2 MAJOR RESEARCH HIGHLIGHTS

2.1 Performance Impact of Parallelisation Strategy

The fact that runtime performance of irregular algorithms is dependent on the structure of input data is commonly accepted in HPC communities. However, there has been little work quantifying the size of this effect.

In [15] I've investigated how the underlying hardware platform affects the performance of different parallelisation strategies, and shown that there is no single best implementation for a given hardware platform, nor for a given input graph. The best implementation is dependent on *both* the hardware and input data.

In [16] I presented a case study on different parallelisation strategies for the pagerank [13] algorithm. The results show that the relative performance of the parallelisation strategies can fluctuate by orders of magnitude across different input datasets. One surprising and noteworthy outcome of these experiments was the viability of edge-based parallelisation. As show in Figure 1

Some CPU-based implementations, such as PGX.D [7], use edge-based parallelisation, but the state-of-the-art in GPU processing frameworks use on vertex-based parallelisation strategies, worklists, or Gather-Apply-Scatter (GAS) approaches. This is, perhaps, due to the assumption that edge-based parallelisation creates too many tasks with not enough computation.

Our experiments show that edge-based parallelisation outperforms these other approaches on the GPU for a majority of the graphs in the public data sets from SNAP [10] and KONECT [9]. As we scale our input graphs to larger sizes, such as RMAT-25 [5]

[1] Properties such as: degree distribution, diameter, clustering coefficient, etc.

No.	Graph	# Vertices	# Edges
1	actor-collaboration	382,219	30,076,166
2	ca-cit-HepPh	28,093	6,296,894
3	discogs_affiliation	2,025,594	10,604,552
4	opsahl-ucsocial	1,899	20,296
5	prosper-loans	89,269	3,330,225
6	web-NotreDame	325,729	1,497,134
7	wikipedia_link_en	12,150,976	378,142,420
8	wikipedia_link_fr	3,023,165	102,382,410
9	zhishi-hudong-internallink	1,984,484	14,869,484
10	R-MAT 24	8,870,942	260,379,520
12	R-MAT 25	17,062m472	523,602,831

Table 1: Details for the input graphs shown in Figure 1, Figure 2, Figure 3, and Figure 4.

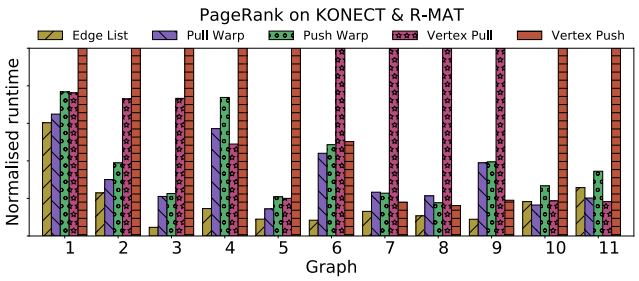

Figure 1: Normalised runtimes of different PageRank implementations on selected KONECT [9] and R-MAT [5] graphs. See Table 1 for the details of the input graphs.

the edge-based parallelisation loses its edge over the other strategies, so it remains an open question how to predict the best performing implementation. These results show that we should not pre-emptively rule out edge-based strategies.

2.2 Systematic Benchmarking

In [16] I also created a sequential workload model for each of the PageRank implementations we have. This model matched the work observed from profiling accurately, but I was unable to create an adequate model for the parallel execution. My difficulty in creating a parallel execution model stems from the data dependence of the parallelisation strategies and the un(der)documented details of the GPU hardware. Due to this lack of an analytical model for parallel speed-up, the models were not useful for predicting actual runtimes. Implementations performing more work often ended up faster than more efficient implementations as a result of the parallelisation.

Instead, I decided to perform series of systematic benchmarks to try and correlate the parallel speed-up with structural properties of the input graph. However, there is no established dataset for such systematic benchmarking. As such, I proposed a novel graph generator [17] that can generate graphs with more control over the properties of the generated graphs.

The idea behind this generator was to use evolutionary computing to generate random graphs the most closely approximates the desired values for each structural property. Evolutionary algorithms are good at optimisation problems that where the constraints correlate with each other, as is the case with structural properties of graphs.

This evolutionary generator works well for the generation of small graphs, but does not scale to graphs of 10,000 or more vertices. At that size the correlation between the properties becomes to complex, as a result it takes 10s of hours to find a single graph that matches the desired criteria. If it finds a solution at all. This is too slow to be useful for any form of systematic benchmarking.

In the absence of a suitable dataset, I settled for benchmarking existing real world graphs and using other techniques to determine the impact of different structural properties. I used graphs from KONECT and SNAP for this benchmarking, running my 15 different implementations of PageRank and BFS on these graphs, as well as the BFS implementations of Gunrock [18] and Lonestar [8]. Figure 2 shows some of the results, illustrating how the runtime of different BFS implementations varies across graph.

This effort resulted in detailed performance data for BFS and pagerank. In total we have timing data for 1,728,090 levels of BFS on 248 graphs, from multiple BFS implementations and multiple starting vertices per graph.

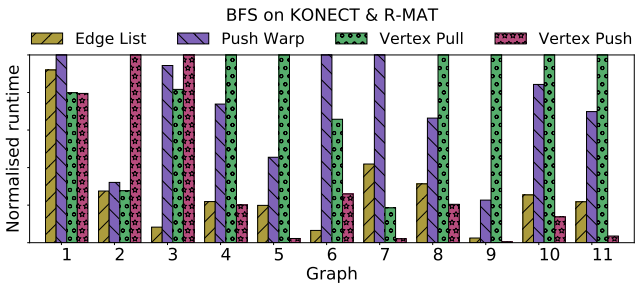

Figure 2: Normalised runtimes of different BFS implementations on selected KONECT [9] and R-MAT [5] graphs. See Table 1 for the details of the input graphs.

2.3 Model Generation Case Study: BFS

Using the corpus of runtime data described above, I further focused on the BFS analysis. As Figure 3 shows, the timings of the individual BFS levels show that the performance varies even more on the individual levels than it does for the overall BFS. Thus, it should be possible to dramatically speed up BFS, if we can predict which of the implementations is fastest for a given level. This allows us to dynamically switch to the predicted/best implementation for each level (this generalises the work done on direction-optimising BFS [2] by Beamer et al., to more than two implementations).

2.3.1 Decision Tree Model. The noisiness of real world datasets rules out evaluating the impact of individual structural properties. Therefore, I used machine learning to model the relation between structural properties of input graphs and the performance of the individual implementations. The method I used were binary decision trees [3]. The two main reasons for using decision trees are: 1) the fact that predictions are cheap to compute, which is important

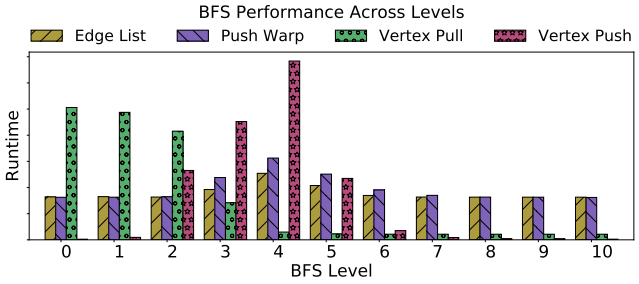

Figure 3: Absolute runtimes at different BFS levels of BFS implementations on a single graph. See Table 1 for the details of the input graphs.

if we want to switch between implementations for different levels of BFS, and 2) the fact that decision trees can output the "predictive power" of the input parameters, which means that we can use the results of trained decision trees to improve and guide our analytical modelling efforts.

2.3.2 Dynamic Switching. The decision trees are accurate enough to provide a significant performance increase. Figure 4 compares our results against the state-of-the-art GPU graph processing framework Gunrock [18] and the slightly older BFS benchmark LonestarGPU [4], 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 2.9× of our theoretical optimum. LonestarGPU manages 21× of optimal. Our model's 1.4× of optimal means that we are, on average, 2× faster than Gunrock.

In the worst case, the time to compute a prediction is about 1% of the time of a single level of BFS, meaning the prediction overhead is negligible. These times do assume we do not perform any data loading/transfer during the algorithm. In other words, if implementations require different in memory data representations, we will have to keep all of these in memory, making a dynamic switching BFS a classic time-space trade-off.

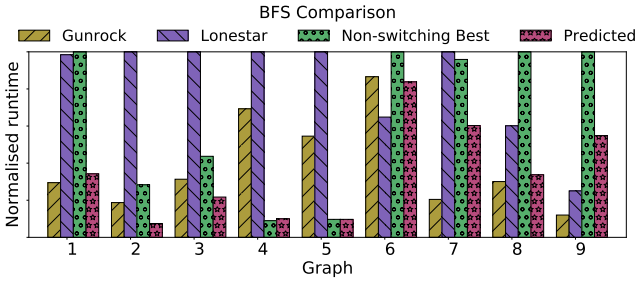

Figure 4: Comparison of normalised runtimes of different BFS implementations, predicted performance, and existing optimised BFS implementations on KONECT [9] graphs. See Table 1 for the details of the input graphs.

2.4 On-going Work

I'm working on turning the ad hoc data processing pipeline used for the analysis and model generation for BFS into an automated pipeline for training predictors for other algorithms. That is, a tool where implementers can provide their own implementations and/or measured runtimes for an algorithm, feed them into this pipeline, and get a decision tree prediction model back out that predicts the best implementation to use for a given input graph.

I'm also investigating how much data we require to train predictors that are accurate enough to be useful without losing generality or overfitting for the training set. This includes establishing rough figures for how many graphs/measurements we need for accurate models, and how varied the input graphs should be to ensure adequate generality of the trained model.

REFERENCES

[1] Graph500. `http://graph500.org`.
[2] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.
[3] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and Regression Trees*. CRC press, 1984.
[4] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.
[5] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.
[6] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *IPDPS*, pages 851–862, 2013.
[7] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi. PGX.D: A fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 58. ACM, 2015.
[8] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 65–76. IEEE, 2009.
[9] J. Kunegis. Konect: The koblenz network collection. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13 Companion, pages 1343–1350, 2013.
[10] J. Leskovec. Stanford Network Analysis Platform (SNAP). *Stanford University*, 2006.
[11] D. Merrill, M. Garland, and A. S. Grimshaw. Scalable GPU graph traversal. In *PPOPP 2012, New Orleans, LA, USA*, pages 117–128, February 2012.
[12] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus topology-driven irregular computations on gpus. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 463–474. IEEE, 2013.
[13] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.
[14] A. Penders. Accelerating Graph Analysis with Heterogeneous Systems. Master's thesis, PDS, EWI, TUDelft, December 2012.
[15] 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.
[16] 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*, pages 528–540. Springer, 2015.
[17] M. Verstraaten, A. L. Varbanescu, and C. de Laat. Synthetic graph generation for systematic exploration of graph structural properties. In *European Conference on Parallel Processing*, pages 557–570. Springer, Cham, 2016.
[18] 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*, page 11. ACM, 2016.