

Quantifying the Performance Impact of Graph Structure on Neighbour Iteration Strategies for PageRank

Merijn Verstraaten^(✉), Ana Lucia Varbanescu, and Cees de Laat

University of Amsterdam, Amsterdam, The Netherlands
{m.e.verstraaten,a.l.varbanescu,delaat}@uva.nl

Abstract. Increases in graph size and analytics complexity have brought graph processing at the forefront of HPC. However, the HPC shift towards manycore accelerators (e.g., GPUs) is not favourable: traditional graph processing is hardly suitable for regular parallelism. Previous work has focused on parallel algorithms for specific graph operations, often using assumptions about the structure of the input graph. However, there has been very little systematic investigation of how strongly a graph’s structure impacts the efficiency of graph operations.

With this article we make propose a step to quantify this impact, focusing on a typical operation: neighbour iteration. We design and implement four strategies for neighbour iteration and introduce a simple model to reason about the expected impact of a graph’s structure on the performance of each strategy. We then use the PageRank algorithm to validate our model. We show that performance is significantly affected by the ability to effectively load-balance the work performed by these strategies across the GPU’s cores.

1 Introduction

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 computations 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, in an attempt to cope with these rapidly increasing workloads. 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, 13].

However, *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.

To simplify working with graphs and to hide the complexity of the underlying platform, many different graph processing systems have been developed [5, 7–9, 12, 17]. Most of these systems provide a clear separation between a simple-to-use front-end, where users are invited to write applications using, most often,

high-level operations, and a highly-optimized back-end, where these operations are translated to execute efficiently on a given platform (i.e., a combination of hardware and software).

Examples of high-level graph operations common in many algorithms (and thus implemented in graph processing systems) are: (a) iteration over all vertices (e.g., in graph statistics); (b) iteration over all edges (e.g., in traversals); (c) iteration over all neighbours of a vertex (e.g., in pagerank); and (d) iteration over all common neighbours of two vertices (e.g., in label propagation). Efficient mapping of such high-level graph operations to lower-level platform-specific primitives is crucial for the overall performance of the application and, consequently, for the adoption of a graph processing system.

In this work, we focus on the performance of graph operations on GPUs, seen as representative massively parallel HPC architectures. In this context, we make the following observations:

1. Speeding up graph processing by using GPUs requires efficient exploitation of the fine-grained parallelism of graph problems [6, 14].
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 [16, 18].
3. The data locality and 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 significantly different memory representation and access patterns [3].

In summary, given a high-level graph processing operation, there are multiple ways we can choose to implement it. Which implementation is the most efficient on a given platform is highly dependent on the structure of the graph being processed [18]. While it is common knowledge that this is the case, there has not yet been a systematic study to quantify how big this effect is and to what extent it correlates with the structure of the input graph. This information has a clear impact on the performance of graph processing backends, as it would allow the system to adapt the implementation to best suit the input data.

However, to enable such adaptation, we must correlate (classes of) graphs with the performance behavior of different primitives on different platforms. To do so, we must: (1) identify possible implementations for the targeted primitive operations, (2) quantify the performance differences per (platform, dataset) pair, and (3) cluster the datasets with similar performance behavior in classes that can be easily characterized.

In this paper we show an example of how this process can be conducted, focusing on the quantification of the observed performance differences for a real application. Specifically, we present four different implementations of neighbour iteration on a GPU and use these to implement the PageRank [15] algorithm.

We measure how the performance of our implementations varies as a result of changing the input graph. Our experimental results demonstrate that the optimal implementation does not just depend on the dataset, but also on the

dataset’s in-memory representation. We also observe similarities between graphs of similar provenance (e.g., road networks show a different performance ranking than web-graphs), but better clustering is necessary to automate this process.

Our contribution in this paper is three-fold: (1) we design and implement four strategies to deal with neighbours iteration as a primitive graph operation, (2) we demonstrate how all these strategies can be used for PageRank, and (3) we quantify the impact these strategies have on the overall performance of PageRank when running on GPUs.

2 Background

In this section we provide a brief introduction on the PageRank algorithm, as well as a short description of the main characteristics of GPUs, the hardware platform we use for this work.

2.1 PageRank

PageRank is an algorithm that calculates rankings of vertices by estimating how important they are. Importance is quantified by the number of edges incoming from other vertices.

A generic PageRank operation works as follows. Given a graph $G = (V, E)$ the PageRank for a vertex $v \in V$ can be calculated as:

$$PR(v) = \frac{1-d}{|V|} + d \sum_{w \in N(v)} \frac{PR(w)}{\rho(w)} \quad (1)$$

Here d is the damping factor, $\rho(w)$ is the outgoing degree of vertex w , and $N(v)$ denotes the neighbourhood of vertex v , that is:

$$w \in N(v) \iff (w, v) \in E$$

This formula is usually implemented iteratively using two steps. In the first step we compute the incoming pagerank from the previous iteration. In the second step, we normalize this new pagerank using the damping factor. These operations are repeated until the difference between iterations falls below a certain threshold or the maximum number of iterations is reached.

2.2 The GPU Architecture

GPUs (Graphical Processing Units) are the most popular accelerators in High Performance Computing (HPC). GPUs are massively parallel processing units, where hundreds of cores, grouped in streaming multiprocessors (SMs), can execute thousands of software threads in parallel. Software threads are grouped into thread blocks, which are scheduled on the SMs. Threads inside the same block can easily communicate and synchronize; communication and synchronization

for threads in different blocks (or for all threads on the platform) are significantly more expensive.

GPUs have a hierarchical memory model with limited, dedicated shared memory per SM and a relatively large global memory. Shared memory is only accessible by threads in the same block, while global memory is accessible to all threads. Typical sizes for global memory are between 1 and 12 GB.

For highly parallel workloads, GPUs outperform sequential units by orders of magnitude. But in cases where not enough parallelism is exposed, or in cases where there are too many dependencies between threads, or where threads diverge, the GPU performance drops significantly. Given the typical characteristics of graph processing applications — low computation-to-communication ratio, poor locality, and irregular, data-driven memory access patterns [2], the efficient use of GPUs for graph processing is not trivial. More importantly, the dataset structure and its characteristics can play a much more important role in the overall performance than in the case of the more flexible multi-core CPUs.

3 Design and Implementation

In this section we present the design and implementation of four different versions of PageRank, and discuss a simple model for estimating their performance.

3.1 Four PageRank Versions

In the iterative implementation of Eq. 1, we (1) sum the incoming pageranks for every vertex, and then (2) update the pagerank for that vertex.

To compute PageRank in parallel, a choice needs to be made on how the application is parallelized. For a massively parallel platform like the GPU, the amount of exposed parallelism should be as large as possible, so there are two simple strategies to choose from: one vertex per thread (i.e., vertex-centric parallelism), or one edge per thread (i.e., edge-centric parallelism).

Next, for the computation itself, the vertex-centric parallelisation requires a second choice, data can be either pushed or pulled from or to a vertex' neighbours. Thus, vertex-centric approaches can be further divided into *push* and *pull*. With *push*, the thread computes the outgoing pagerank of its vertex and sums that value to all the vertex' neighbours. With *pull*, the thread computes the outgoing rank of the vertex' neighbours and sums them to itself.

Algorithms 1, 2 and 3 show pseudocode implementations of the push, pull and edge-based implementations, respectively. For push and pull these kernels are executed once per vertex, for edge-based the kernel is executed once per edge. A pseudocode implementation of the rank consolidation kernel can be found in Algorithm 4.

We use the following representations. The edge based kernel uses one edge array (origin + destination vertices) and an offset array to compute degrees, resulting in $2 \cdot |E| + |V|$ ints space usage. The push and pull based kernels use Compressed Sparse Row (CSR) and reversed CSR, respectively,

using $|E| + |V|$ ints of space. The pull kernel uses an additional offset array to compute neighbour degrees, using an extra $|V|$ ints of space.

Algorithm 1. Push Vertex-based Update

```

function VERTEXPUSH(v)
  if v.degree  $\neq$  0 then
    outgoingRank  $\leftarrow$   $\frac{v.pagerank}{v.degree}$ 
  end if
  for nbr  $\in$  v.neighbours do
    nbr.newRank.atomicAdd(outgoingRank)
  end for
end function

```

Algorithm 2. Pull Vertex-based Update

```

function VERTEXPULL(v)
  newRank = 0
  for nbr  $\in$  v.neighbours do
    newRank  $+=$   $\frac{nbr.pagerank}{nbr.degree}$ 
  end for
  v.newRank  $\leftarrow$  newRank
end function

```

Algorithm 3. Edge-based Update

```

function EDGEBASED(edge)
  origin  $\leftarrow$  edge.origin
  dest  $\leftarrow$  edge.destination
  outgoingRank  $\leftarrow$   $\frac{origin.pagerank}{origin.degree}$ 
  dest.newRank.atomicAdd(outgoingRank)
end function

```

Algorithm 4. Consolidate kernel

```

function CONSOLIDATERANK(v)
  newRank  $\leftarrow$   $\frac{1-damping}{graphSize} +$ 
    (damping  $\cdot$  v.newRank)
  diff  $\leftarrow$  abs(newRank - v.pagerank)
  globalDiff.atomicAdd(diff)
  v.pagerank  $\leftarrow$  newRank
  v.newRank  $\leftarrow$  0
end function

```

Looking at the kernel for pull vertex-based computation, we observe that it is performing more work than strictly necessary. Computing the incoming rank from every neighbour means that vertices that share neighbours unnecessarily replicate work of dividing the rank. We could simply move this division into the consolidation kernel, performing this computation once per vertex. This requires us to use a different consolidation kernel for the last iteration to obtain the correct results, but this is not particularly difficult. Pseudocode for the modified pull kernel (entitled NoDiv) can be seen in Algorithm 5 and the corresponding consolidation kernel in Algorithm 6.

Algorithm 5. NoDiv: Pull Vertex-Based

```

function VERTEXPULLNO DIV(v)
  newRank  $\leftarrow$  0
  for nbr  $\in$  v.neighbors do
    newRank  $+=$  nbr.pagerank
  end for
  v.newRank  $\leftarrow$  newRank
end function

```

Algorithm 6. NoDiv: Consolidate kernel

```

function CONSOLIDATERANKNO DIV(v)
  newRank  $\leftarrow$   $\frac{1-damping}{graphSize} +$ 
    (damping  $\cdot$  v.newRank)
  diff  $\leftarrow$  abs(newRank - v.pagerank)
  globalDiff.atomicAdd(diff)
  v.pagerank  $\leftarrow$   $\frac{newRank}{v.degree}$ 
  v.newRank  $\leftarrow$  0
end function

```

3.2 Estimating Performance

The above kernels show that the computational workload of pagerank is negligible. Like for many other graph algorithms, most of the workload comes from reading and writing memory. To achieve our goal of correlating algorithm

performance with input data, we need a performance model for our primitives. Our performance model only considers global memory accesses and global atomic operations to reason about the relative work complexity of the different kernels.

For all models, let T_{read} be the cost of a random global read, T_{write} the cost of a random global write, and T_{atom} the cost of a global atomic add operation. For now, we ignore the variability of atomic operation contention and cache effects, in an attempt to only rank the different versions of the algorithm, and *not* predict accurate execution times.

We see in Algorithm 1 that every thread performs 3 reads (2 to compute the degree and 1 to read its pagerank), followed by d atomic addition operations, where d is the degree of that vertex. The number of operations performed by push thus boil down to:

$$T_{push} = \sum_{v \in V} (3 * T_{read} + d_v * T_{atom}) = 3 * |V| * T_{read} + |E| * T_{atom}$$

In Algorithm 2 we see that the pull kernel performs 3 reads for each neighbour of its vertex, and then performs a non-atomic write to store the new result. The total operations performed by pull thus boil down to:

$$T_{pull} = \sum_{v \in V} (3 * d_v * T_{read} + T_{write}) = 3 * |E| * T_{read} + |V| * T_{write}$$

The kernel in Algorithm 3 uses one thread per edge, and each thread performs 3 reads, 2 to compute the degree and 1 to read the pagerank, it then performs an atomic addition to store the result, resulting in:

$$T_{edge} = \sum_{e \in E} (3 * T_{read} + T_{atom}) = 3 * |E| * T_{read} + |E| * T_{atom}$$

The pagerank consolidation kernel is the same for each of the above kernels, performing 2 reads, one for the new incoming rank value and one for the old pagerank value, followed by an atomic addition and 2 writes to store the new pagerank and reset the incoming rank:

$$T_{con} = \sum_{v \in V} (T_{read} + 2 * T_{write} + T_{atom}) = 2 * |V| * T_{read} + 2 * |V| * T_{write} + |V| * T_{atom}$$

The performance model for the optimised pull-based kernel (i.e., NoDiv, Algorithm 5) is:

$$T_{NoDiv} = \sum_{v \in V} (d_v * T_{read} + T_{write}) = |E| * T_{read} + |V| * T_{write}$$

The corresponding consolidation needs a slight update, according to Algorithm 6:

$$\begin{aligned} T_{con.NoDiv} &= \sum_{v \in V} (4 * T_{read} + 2 * T_{write} + T_{atom}) \\ &= 4 * |V| * T_{read} + 2 * |V| * T_{write} + |V| * T_{atom} \end{aligned}$$

Summarizing, these are the performance models for a single pagerank iteration, running sequentially:

$$\begin{aligned}
T_{push} &= 5 * |V| * T_{read} + 2 * |V| * T_{write} + (|V| + |E|) * T_{atom} \\
T_{pull} &= (3 * |E| + 2 * |V|) * T_{read} + 3 * |V| * T_{write} + |V| * T_{atom} \\
T_{NoDiv} &= (|E| + 4 * |V|) * T_{read} + 3 * |V| * T_{write} + |V| * T_{atom} \\
T_{edge} &= (3 * |E| + 2 * |V|) * T_{read} + 2 * |V| * T_{write} + (|V| + |E|) * T_{atom}
\end{aligned}$$

In most graphs, even sparse ones, we can expect $|E|$ to be at least as big as $|V|$ and usually significantly bigger. Given this assumption we can see that the edge-based implementation performs both the most reads and atomic additions. The pull-based implementation performs strictly less work than the edge-based one, as it reduces the number of reads by $3 * (|E| - |V|)$. The pull-based implementation reduces the number of atomic operations required by increasing the number of write operations. The optimised NoDiv version further reduces the number of reads done by $2 * (|E| - |V|)$.

3.3 Parallel Performance

A naive reading of the performance models would conclude that the edge-based version is always slower and the only implementation worth considering are push and pull. However, in practice the comparison is not as straightforward. When running in parallel, on the GPU, the performance depends on the number of threads, chosen architecture (number of SMs), and scheduling. GPUs schedule threads in groups, usually called warps, and every thread in the warp executes the same instruction.

Divergent loops within a warp result in idle cores while executing that warp; the performance of the entire warp is thus limited to the slowest thread. This means that processing vertices of differing degrees within the same warp leads to efficiency loss due to diverging loops in the push and pull kernels. The edge-based version does not suffer from divergence and all of the GPU cores are always utilised. Therefore, the choice between push/pull updates and edge-based updates is a trade-off between performing extra work for better workload balance.

The question we need to answer is: At what point does the intra-warp workload imbalance start to outweigh the extra work performed by the additional operations performed by the edge based implementation? In this work, we answer this question empirically, and demonstrate that the degree distribution plays an important role in this decision.

4 Experimental Evaluation

With the simple performance models introduced in the previous section, we expect that push and pull perform best on graphs that have a (near) constant degree, as this results in very good/perfect workload balance between all threads within a warp. Correspondingly, we expect both to perform worse for graphs that have large variation in degree.

In this section we empirically validate this hypothesis and measure the trade-off between the extra work done by the edge based version against the impact

of workload imbalance for the push and pull versions. To do so, we ran all four versions of PageRank (see Sect. 3) on multiple datasets, both real world datasets from SNAP [11] and artificially generated graphs.

4.1 Experimental Setup

For running PageRank, we used a damping factor of 0.85. We ran the algorithm for 30 iterations to avoid convergence differences. The results presented here consist of the time the PageRank computation took, averaged over 30 runs, excluding data transfers to and from the GPU. We performed these measurements on an NVIDIA K20 (an HPC-oriented GPU card, with lower memory bandwidth, but larger global memory). We used version 5.5 of the CUDA toolkit.

In addition to the variations described in Sect. 3 we also implemented alternate versions of the push and pull kernels, based on the work of Hong, et al.; which showed a technique for achieving smoother load-balancing for vertex-centric programming on the GPU, leading to speed-ups up to 16x for certain graphs. [10]

For the edge-based implementation, we implemented both a struct-of-arrays and array-of-structs implementation of our edge data structure. Array-of-structs is a common optimisation technique on the CPU, but it is not clear whether the same technique is an optimisation on the GPU, and we aimed to determine this empirically.

To summarise we have 8 versions: edge-based using array-of-structs, edge-based using struct-of-arrays, push, pull, optimised pull, plus warp-optimised versions of the latter three. For the warp versions we have tried warp sizes 1, 2, 4, 8, 16, 32, and 64 with chunk sizes ranging from 1 to 10 times the warp size. All these versions are available online at <https://github.com/merijn/GPU-benchmarks>.

We have selected 19 datasets from several different classes of graphs from the SNAP [11] repository. These include citation, collaboration, social, computer, and road networks. The characteristics of the datasets are presented in Table 1.

4.2 Results

In Fig. 1, we show the normalised results of our experiments, meaning that the worst performing implementation of PageRank for each graph is plotted as 1, and all the others are fractions of the worst performing variant (i.e., lower is better, and the lower the bar, the higher the performance gap). For readability reasons we filtered out all the warp implementations of push and pull that did *not* perform faster than any other implementation.

Our initial hypothesis of push and pull performing best on graphs with constant degree is confirmed by the performance measured on our artificial graphs with fixed degrees. Both push and the optimised pull win on all but one of these. We also see them performing well on the road networks. This is not surprising, because the road networks have fairly little variation in terms of the degree of nodes (the highest degree is 6). On the other hand, star presents a worst-case scenario for push and pull, having a completely imbalanced workload. As confirmed by the large performance gap in the results for that graph.

Table 1. Our 7 synthetic graphs, followed by the 12 real world graphs from SNAP.

No	Graph	$ V $	$ E $	Description
1	chain_1000000	1,000,000	2,000,000	Bidirectional chain
2	star_1000000	1,000,000	2,000,000	Star
3	degree4_1000	999,999	4,000,000	Two dimensional mesh
4	degree6_100	999,999	6,000,000	Three dimensional mesh
5	degree.5_16	1,048,575	10,485,760	Constant out-degree 10
6	degree.10_4	1,048,575	20,971,520	Constant out-degree 20
7	degree.20_2	1,048,575	41,943,040	Constant out-degree 40
8	as-Skitter	1,696,415	22,190,596	Internet topology graph
9	cit-Patents	3,774,768	16,518,948	Citation network among US Patents
10	email-EuAll	265,214	420,045	Email from a EU research institution
11	Facebook	4,039	176,468	Social circles from Facebook
12	Gplus	107,614	13,673,453	Social circles from Google+
13	roadNet-CA	1,965,206	5,533,214	Road network CA
14	roadNet-TX	1,379,917	3,843,320	Road network TX
15	soc-LiveJournal1	4,847,571	68,993,773	LiveJournal online social network
16	Twitter	81,306	1,768,149	Social circles from Twitter
17	web-BerkStan	685,230	7,600,595	Web graph of Berkeley and Stanford
18	web-Google	875,713	5,105,039	Web graph from Google
19	wiki-Talk	2,394,385	5,021,410	Wikipedia talk comm. network

We note that even under ideal circumstances for push and pull, the edge-based implementation is not far behind in terms of performance, despite performing substantially more work than push and pull.

We also observe that there is very little difference between the two edge-based implementation. Surprisingly, these results show that the array-of-structs optimisation used to exploit cache locality on the CPU has no significant impact on the algorithm's performance on the GPU. In fact, it appears to be marginally slower on all graphs.

Another, perhaps surprising, result is that the warp versions of push and pull inspired by [10] almost never win in terms of performance. The trade-off made by the warp-optimisation is that it tries to smooth the load-balancing by performing more work than the pure vertex-centric code. As a result it is somewhere between edge-based and the push or pull based version. As a result it load-balances less well than edge-based, but has more overhead than push/pull for the ideal constant degree graphs. As such, its performance appears to combine the worst of both worlds.

4.3 Sorted Graphs

Vertices within a warp having different degrees result in workload-imbalance for the push and pull algorithms. Sorting the vertices within a graph by their

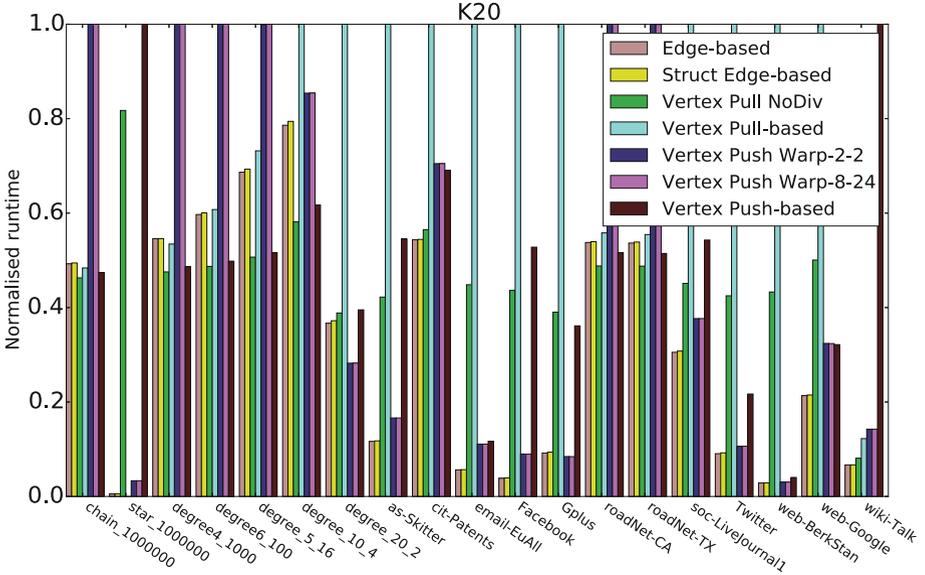


Fig. 1. Normalized performance of the PageRank implementations for our 19 graphs running on NVIDIA K20. The worst performing implementation is used for normalization - i.e., lower is better, and the lower the value, the higher the gap to the worst performing version (Color figure online).

degree would ensure that all vertices are neighbored by vertices of similar degree in the Compressed Sparse Row (CSR) representation, reducing the workload imbalance.

What we found is that sorting the vertices changes the caching and contention patterns change. The result is that in about half the cases sorting the graph vertices had no impact on the performance. In the half where it did have an impact, the results vary. For example, in Fig. 2a we see that the sorted graphs result in a substantially slower push and pull performance. On the other hand, in Fig. 2b we note an improvement for these same implementations.

Overall, our results demonstrate that different implementations of basic graph operations do depend on the structure of the input graph, as seen by the significant fluctuations in the performance of three out of four implementations on different graphs. Additionally, they illustrate that effective load-balancing is the most important feature to obtain good performance from the GPU.

Our experiments with sorting demonstrate that fixing the load-balancing for push and pull is not as straightforward as simply sorting the vertices within a graph by their degree. This due to changes in caching and contention patterns. With the exception of the cit-Patents results shown in Figure 2b, the sorting did not impact which algorithm was the best performing for a specific graph.

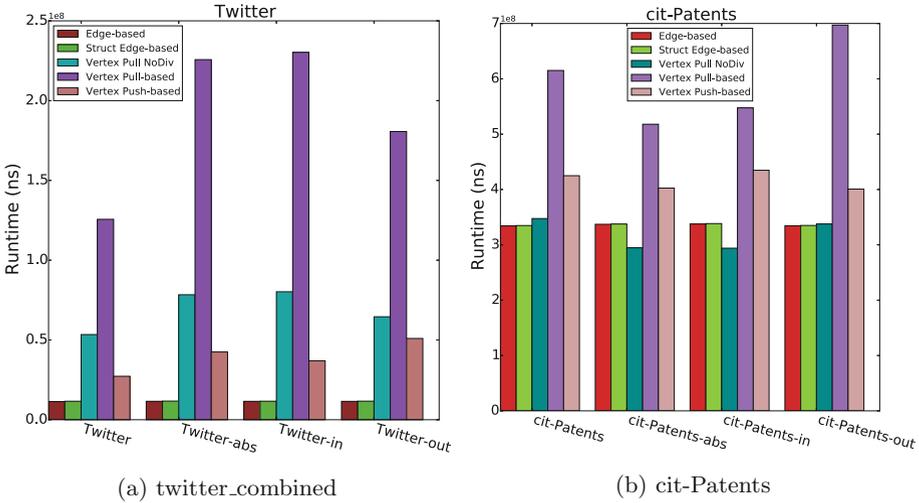


Fig. 2. Impact of sorting vertices by degree on PageRank performance (Color figure online).

What remains to be seen is whether this apparent superiority of edge-based neighbour iteration is an artifact of the PageRank algorithm we used for evaluation, or whether this holds across different algorithms.

5 Related Work

Multiple studies already demonstrate the impact of different implementations of the same graph processing algorithm on GPUs [3, 10, 13]. In most cases, however, such research focuses on different design, implementation, and tuning options which can be applied to *favour* the (hardware) platform, without paying attention to the datasets. In this work, we focus on the performance impact that graphs have on the *efficiency* of these optimizations, determining whether an unfriendly graph can render a given optimization useless.

Another line of research focuses on applications designed for a specific class of algorithms — e.g., efficient traversing of road networks [4] — where the properties of the graphs are taken into account when constructing the algorithm. However, this approach lacks generality, as such algorithms will simply not work for other classes of graphs. We instead aim to rank generic graph-processing solutions by their performance on different types of graphs.

Finally, several studies have observed the impact of graphs on the performance achieved by various graph processing systems [5, 7–9, 12, 17, 18], yet most of them have analyzed this dependency at the level of the full algorithm, not at the level of its basic operations. In our work, we focus on a systematic, fine-grained analysis, performed at the level of basic graph operations. We believe this bottom-up approach is key to providing a performance-aware design for new graph processing systems.

6 Conclusion

With the increased diversity of hardware architectures, different algorithms and implementations are being developed for regular graph operations. In this paper, we have studied four different strategies to implement neighbour iteration, and demonstrated their usability in PageRank. Further, focusing on the performance of PageRank on 19 different datasets, we demonstrated that different strategies have different performance behavior on different datasets.

In the near future, we will work to validate and improve our performance models. We plan to expand our experiments to other algorithms to investigate whether the apparent superiority of edge-based neighbour iteration is an artifact of PageRank, or an intrinsic property of neighbour iteration. Additionally we plan to expand this work to other graph processing primitives, such as common neighbour iteration, as found in triangle counting/listing.

References

1. Graph500. <http://graph500.org>
2. Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J.W.: Challenges in parallel graph processing. *Parallel Process. Lett.* **17**, 5–20 (2007)
3. Burtscher, M., Nasre, R., Pingali, K.: A quantitative study of irregular programs on GPUs. In: 2012 IEEE International Symposium on Workload Characterization (IISWC), pp. 141–151. IEEE (2012)
4. Delling, D., Kobitzsch, M., Werneck, R.F.: Customizing driving directions with GPUs. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014 Parallel Processing. LNCS, vol. 8632, pp. 728–739. Springer, Heidelberg (2014)
5. Elser, B., Montresor, A.: An evaluation study of bigdata frameworks for graph processing. In: *Big Data* (2013)
6. Gharaibeh, A., Costa, L.B., Santos-Neto, E., Ripeanu, M.: On graphs, GPUs, and blind dating: a workload to processor matchmaking quest. In: IPDPS, pp. 851–862 (2013)
7. Guo, Y., Biczak, M., Varbanescu, A.L., Iosup, A., Martella, C., Willke, T.L.: How Well do graph-processing platforms perform? an empirical performance evaluation and analysis. In: IPDPS (2014)
8. Guo, Y., Varbanescu, A.L., Iosup, A., Epema, D.: An empirical performance evaluation of GPU-enabled graph-processing systems. In: CCGrid 2015 (2015)
9. Han, M., Daudjee, K., Ammar, K., Ozsu, M.T., Wang, X., Jin, T.: An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endowment* **7**, 1047–1058 (2014)
10. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating CUDA graph algorithms at maximum warp. In: *ACM SIGPLAN Notices*. vol. 46, pp. 267–276. ACM (2011)
11. Leskovec, J.: Stanford Network Analysis Platform (SNAP). Stanford University (2006)
12. Lu, Y., Cheng, J., Yan, D., Wu, H.: Large-scale distributed graph computing systems: an experimental evaluation. *Proc. VLDB Endowment* **8**, 281–292 (2014)
13. Merrill, D., Garland, M., Grimshaw, A.S.: Scalable GPU graph traversal. In: PPOPP 2012, New Orleans, LA, USA. pp. 117–128, February 2012

14. Nasre, R., Burtscher, M., Pingali, K.: Data-driven versus topology-driven irregular computations on gpus. In: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 463–474. IEEE (2013)
15. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: bringing order to the web. Technical report 1999–66, Stanford InfoLab, previous number = SIDL-WP-1999-0120, November 1999. <http://ilpubs.stanford.edu:8090/422/>
16. Penders, A.: Accelerating graph analysis with heterogeneous systems. Master’s thesis, PDS, EWI, TUDelft, December 2012
17. Satish, N., Sundaram, N., Patwary, M.A., Seo, J., Park, J., Hassaan, M.A., Sengupta, S., Yin, Z., Dubey, P.: Navigating the maze of graph analytics frameworks using massive graph datasets. In: SIGMOD (2014)
18. Varbanescu, A.L., Verstraaten, M., Penders, A., Sips, H., de Laat, C.: Can portability improve performance? an empirical study of parallel graph analytics. In: ICPE 2015 (2015)