# Can Portability Improve Performance? An Empirical Study of Parallel Graph Analytics

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

Ate Penders, Alexandru Iosup,
Henk Sips
Delft University of Technology
The Netherlands
{a.b.penders, a.iosup, h.j.sips}@tudelft.nl

## ABSTRACT

Due to increasingly large datasets, graph analytics—traversals, all-pairs shortest path computations, centrality measures, etc.—are becoming the focus of high-performance computing (HPC). Because HPC is currently dominated by many-core architectures (both CPUs and GPUs), new graph processing solutions have to be defined to efficiently use such computing resources. Prior work focuses on platform-specific performance studies and on platform-specific algorithm development, successfully proving that algorithms highly tuned to GPUs *or* multi-core CPUs can provide high performance graph analytics. However, the portability of such algorithms remains an important concern for many users, especially the many companies without the resources to invest in HPC or concerned about lock-in in single-use parallel techniques.

In this work, we investigate the functional portability and performance of graph analytics algorithms. We conduct an empirical study measuring the performance of 3 graph analytics algorithms (a single code implemented in OpenCL and targeted at many-core CPUs *and* GPUs), on 3 different platforms, using 11 real-world and synthetic datasets. Our results show that the code is functionally portable, that is, the applications can run unchanged on both CPUs and GPUs. The large variation in their observed performance indicates that portability is necessary not only for productivity, but, surprisingly, also for performance. We conjecture that the impact of datasets on performance is too high to allow platform-specific algorithms to outperform the portable algorithms by large margins, *in all cases*. Our conclusion is that portable parallel graph analytics is feasible without significant performance loss, and provides a productive alternative to the expensive trial-and-error selection of one algorithm for each (graph,platform) pair.

## Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; D.1.3 [**Software**]: Concurrent Programming

## Keywords

Parallel Graph Analytics; Portability; OpenCL; GPU; Multi-core processors; Many-core processors


*ICPE'15,* Jan. 31–Feb. 4, 2015, Austin, Texas, USA.

http://dx.doi.org/10.1145/2668930.2688042.

## 1. INTRODUCTION

While graph processing is at the core of a large variety of applications, from path finding in maps and networks to bioinformatics, from circuit floor planning to text and speech analysis, it is probably large social and professional networks, like Facebook and LinkedIn, that have brought graph analytics back into the spotlight. Given the extreme scales of the datasets that need to be analyzed, as well as the more extensive analysis that needs to be performed, graph analytics has become a high-performance computing (HPC) concern. This trend is probably best proven by the intense activity and fast changes happening in the Graph500[1] ranking, as well as in the adoption of graph traversals as important benchmarks [6] and drivers for irregular algorithms programming paradigms [26].

At the same time, the state-of-the-art in high performance computing is massive parallel processing, backed up by a large variety of parallel platforms ranging from graphical processing units (GPUs) to multi-core CPUs and Xeon Phi. Because traditional graph processing algorithms are known for their parallelism-unfriendly features - data-dependency, irregular processing, bad spatial and temporal locality [1] - a lot of work has focused on developing GPU-specific [17, 29, 5, 24], multi-core CPU-specific [2], or even vendor-specific [7, 27] algorithms. *None* of these algorithms is portable between different families of platforms. This platform lock-in is undesirable for users that want to make the best use of their resources for longer than a couple months - e.g., small and medium enterprises (SMEs). Moreover, as graph processing workloads increase in size, single-node platforms quickly become unfeasible, and multi-node solutions such as clusters and clouds must be used to tackle "big-data" graphs [35]. For productivity reasons, such platforms must rely on some degree of software stability. We argue in this work that code portability is one way towards this stability, and we show that the performance of such a scenario is, for the specific case of graph analytics, sufficient.

Programming models such as OpenCL[2] allow users to write functionally portable programs for massively parallel architectures with controlled performance losses (if any) compared to native solutions such as CUDA or OpenMP [15, 30]. In our study, we use OpenCL to implement three well-known graph analytics workloads - breadth-first search traversal (BFS), all-pairs shortest path computation (APSP), and betweenness centrality calculation (BC). We test the performance of these 3 algorithms on 3 platforms using 11 real-life and synthetic datasets, and we observe a large diversity in our results: no (platform, algorithm) pair wins for all datasets, and no (graph, platform) pair is superior for all algorithms. In other words, given a graph and an algorithm, one needs to empirically determine which platform is the best performing one. Having port-

[1] http://www.graph500.org
[2] https://www.khronos.org/opencl/

able code enables this choice on any graph and any platform, and it is agnostic to platform heterogeneity in multi-node systems.

The contribution of this work is three-fold:

1. We present three parallel graph analytics workloads implemented in OpenCL (Section 3) and prove their functional portability.

2. We provide empirical evidence that performance depends significantly on *three factors:* datasets, algorithms and data structures, and platforms (Section 4).

3. We show empirical evidence that portability can improve performance (Section 4) and discuss the operational impacts, for real-life users, of the trade-off between performance and portability.

# 2. BACKGROUND

In this section, we present the two families of hardware used for this study and OpenCL, our programming model of choice.

## 2.1 The architectures: CPUs vs. GPUs

In 2007, the bundle of parallelism and multiple cores has been proposed as *the* solution to increase performance. Since then, a large variety of architectures - including Cell/B.E., the GPGPU many-cores, multi-core and many-core CPUs, and the Fusion architectures - have been proposed, discussed, benchmarked, upgraded, and/or dismissed. Currently, the HPC community focuses mostly on multi-core CPUs and many-core accelerators (GPUs).

Multi-core CPUs are architectures that combine a few homogeneous cores (curently, 6 to 8, but slowly increasing over the years), additionally augmented with hardware multi-threading. These cores are complex architectures, much like the traditional CPUs. The memory system uses relatively deep cache hierarchies (2-3 levels), with both per-core and shared caches, as well as shared global memory. In terms of parallelism, multi-cores have little restrictions, allowing both symmetric and asymmetric multi-threading applications. Fine-grain parallelism is exploited by vector units. Finally, multi-cores are stand-alone systems, and the mapping of threads to cores is typically delegated to an operating system and/or a runtime system.

By contrast, Graphics Processing Units (GPUs) have hundreds to thousands simple processing elements (called "threads" by NVIDIA and "processing units" (PUs) by OpenCL), which can be further grouped in core-like entities (called "streaming multi-processors" by NVIDIA and "compute units" by OpenCL). The memory model is based on a large shared global memory, augmented with private and distributed local memories. Using a relaxed memory model, GPUs require the users to address memory consistency issues. In terms of parallelism, GPUs focus on massively data-parallel applications. Thus, programmers write the operations one thread needs to perform, and launch a sufficient (typically large) number of threads. Thread mapping and scheduling are done very efficiently by the hardware. GPUs are typically used as accelerators, to process compute-intensive tasks offloaded to them by a "host" device.

To summarize, GPUs are aiming at speeding-up massively data parallel applications, while multi-core CPUs are typically a better option for more complex parallel processing patterns (e.g., asymmetric multithreading or pipelining). As graph processing seems to be neither of the two [1], we use both CPUs and GPUs in our evaluation, in the (indirect) search of a clear winner in terms of performance.

## 2.2 The programming model: OpenCL

Proposed in 2008 (by the KHRONOS group), OpenCL aims to tackle the platform diversity problem by offering a common hardware model for all multi- and many-core platforms. The user programs this "virtual" platform, and the resulting source code is portable on any OpenCL compliant platform[3].

An OpenCL program has two types of code: the kernel(s), which are the basic unit of computation to be executed on one or more OpenCL devices, and the host program, which is executed on the host. A host program defines the context for the kernels and manages their execution. Note that the device and the host - and consequently the kernel and the host program - have separate memory spaces: any communication between them needs to be explicit.

A compute kernel can be thought of as similar to a C function which specifies the computation that each thread (i.e., *work-item* in OpenCL terminology) needs to perform. The threads can be grouped in *work-groups*, which allow for synchronization and shared memory. On the host side, the programmer needs to write the host code, which, besides application-specific initialization, needs to setup the OpenCL context, choose a running configuration for the kernel (i.e., the number of work-items and the size of the work-groups), copy data to the device (if needed), launch the kernels, and copy the results of the kernel.

The strongest point of OpenCL is its functional portability, a result of using a *common platform model* as a virtual middleware. This separates the design and implementation concerns: *programmers* are only concerned with designing a parallel application for the given platform model, while it is the responsibility of the *hardware vendors* to provide good OpenCL drivers to map the platform to real hardware. Portability is the main reason for which we chose OpenCL for our work: we want to avoid comparing different algorithms and/or implementations for CPUs and GPUs.

# 3. APPLICATIONS

We study three common graph processing algorithms: breadth first search (BFS), all-pairs shortest paths computation (APSP), and betweenness centrality computation (BC). When implementing these algorithms, we opted for a controlled parallelism increase: each algorithm attempts to add extra parallelism on top of the previous one. Thus, APSP uses BFS, and BC uses both APSP and BFS. While this is not the best implementation in terms of performance, it is definitely a valid one (i.e., it leads to correct results), and allows us to better isolate the impact of algorithm parallelism in the overall performance. We will address the performance concerns in more detail in Section 4.

We further note that the graph representation we have chosen is edge-based - i.e., the graph is represented as a list of edges (similar to the representation found in SNAP[4]. Moreover, our implementation does not attempt to reconstruct the graph, nor to transform it to a different representation (such as adjacency lists or adjacency matrix). Instead, we use the edge-based representation directly in our parallel BFS, which propagates further into APSP and BC.

## 3.1 Breadth First Search (BFS)

A BFS traversal explores a graph level by level. Given a graph $G = (V, E)$, with $V$ its collection of vertices and $E$ its collection of edges, and a source vertex $s$ (considered as the only vertex on

[3]Currently (December 2014), these devices have hardware drivers and compiler back-ends: AMD, NVIDIA, and ARM GPUs, AMD's multi-core CPUs and APUs, Intel's CPUs and Intel's Xeon Phi, the Cell/B.E, and Altera's FPGAs

[4]`http://snap.stanford.edu/snap/`

level 0), BFS systematically explores edges outgoing from vertices at level $i$ and places all their destination vertices on level $i+1$, *if* these vertices have not been already discovered at prior levels (i.e., the algorithm has to distinguish *discovered* and *undiscovered* vertices to prevent infinite loops).

In a BFS that accepts an edge list as input, an iteration over the entire set of edges is required for each iteration. By a simple check on the source vertex of an edge, the algorithm can determine which edges to traverse, hence which destination vertices to place in the next level of the resulting tree.

Our parallel BFS works by dividing the edge list into sub-lists, which are processed in parallel (see the kernel listed in Algorithm 3): each thread will traverse its own sub-list in every iteration. Synchronization between levels is mandatory to insure a full exploration of the current level before starting the next one.

When mapping this parallel kernel to OpenCL, each thread is mapped to an work-item. As global synchronization is necessary, we implement a two-layer barrier structure: first at work-group level (provided by OpenCL), then between work-groups (implemented in-house). This solution limits the synchronization penalty - see [25], Chapter 3 for more details, or check Algorithms 1 and 3 in the Appendix.

## 3.2 All Pairs Shortest Paths (APSP)

The problem of finding the minimal distance between all pairs of nodes in a graph is called All-Pair Shortest Paths (APSP). The APSP algorithm gets a graph $G = (V, E)$ and computes, for each pair of vertices $(u, v) \in V$, the shortest path from $u$ to $v$. The length of a path is the sum of its constituent edges.

There are multiple algorithms for computing APSP. For example, Johnson's algorithm [16] uses the Bellman-Form algorithm [3, 10] to remove negative edges and then applies Dijkstra's algorithm [23, 32] for finding the shortest paths. Another approach is based on dynamic programming, as shown by the Floyd-Warshall algorithm [9]. This algorithm gradually builds the full paths by choosing a next best step in each iteration.

In this work, we see APSP as a collection of $N = |V|$ shortest path problems, where $N$ is the number of vertices in the graph. By systematically performing a BFS traversal (see Algorithm 2 in the Appendix) for each shortest path problem (using different source vertices), we solve the full query. Note that all the BFS traversals are kept independent (i.e., we do not implement any optimizations) in order to increase the parallelism level of the APSP algorithm as compared with the BFS one.

To parallelize APSP, one could adopt three strategies: (1) execute each BFS in parallel, and loop over all $N$ vertices sequentially, (2) execute each BFS sequentially (i.e., in a single thread), and run all $N$ instances of BFS concurrently, or (3) a mixed approach, where each BFS is parallelized itself, and more BFS instances are executed in parallel. We choose option (3), as it maps best to the two-layer parallelism of OpenCL: each BFS is parallelized per work-group, and concurrent work-groups run multiple BFS's in parallel. To achieve the full APSP, each workgroup has to sequentially iterate over an own sub-list of BFS sources (for more details, please check Algorithm 4 in the Appendix).

Note that this design choice also reduces the complexity of our BFS synchronization, replaced now by the built-in work-group synchronization. However, statically assigning groups of vertices to work-groups can lead to high imbalance between work-groups, potentially leading to low platform utilization. Improving this point is mandatory for a better performing BFS, but less important for our parallelism analysis (a complete analysis of all these options is presented in [25], Chapter 4).

## 3.3 Betweenness Centrality (BC)

Centrality analysis provides detailed information about the impact of individual vertices in a graph structure, by measuring the "influence" a vertex has on the connectivity of the graph.

A widely used BC algorithm, by Freeman [11], searches all the shortest paths between any two vertices and assigns a degree of betweenness (between 0 and 1) to the intermediate vertices. An intermediate vertex has a degree of 1 if and only if all the shortest paths between two other vertices pass through it, and 0 if no shortest path passes through it. The BC index of a vertex $v$ is the sum of degrees of betweenness for all pairs of vertices (see equation 1) whose shortest connection passes through $v$. Here, $\sigma_{st}$ denotes the total number of shortest paths between $s$ and $t$, and $\sigma_{st}(v)$ the count of shortest paths that pass through $v$.

$$BC(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (1)$$

To calculate the BC of a graph, we compute the number of shortest paths between pairs of vertices, remember the vertices on each of these paths, and determine the ratio of shortest paths passing through each vertex in the graph. This procedure is repeated for all pairs of vertices in the graph, with $(s, t) = (t, s)$, and the pair-dependencies are accumulated per vertex.

When implementing this algorithm, we distinguish three steps: (i) a traversal to get the total number of shortest paths for a pair of vertices, (ii) computing all ratios of (shortest) paths passing each vertex for a pair of vertices, and (iii) adding the derived pair-dependencies (i.e., the ratios of shortest paths) for each vertex to its BC value. This sequence of steps is repeated for computing pair-dependencies of all vertices for the different pairs of vertices, allowing us to use the same strategy as for APSP: we iterate (partially in parallel) over steps (i), (ii) and (iii) for each vertex.

A BFS is used for step (i), to derive the total number of shortest paths for a pair of vertices. Based on a technique presented in [4], we also use a reverse BFS traversal (i.e., a *bottom-up BFS*) to accumulate the ratios of the counts for the intermediate vertices in step (ii). At each step of the bottom-up BFS, the score $\delta_{s\bullet}(v)$ of vertex $v$ is computed as the accumulated score of all individual children of $v$, as seen in Equation 2. Finally, we retrieve the pair-dependency values assigned to the vertices in the previous step and add them all to compute the final BC value of each vertex (step (iii)). For more details, on this implementation, please refer to [25], Chapter 5.

$$\delta_{s\bullet}(v) = \sum_{c:v \in Parent(c)} \frac{\sigma_{sv}}{\sigma_{sc}} \cdot (\delta_{s\bullet} + 1) \quad (2)$$

To summarize, our implementation uses one APSP that includes $2 \cdot N$ BFS traversals, allowing us to build upon the existing implementations of these kernels (for implementation details, Algorithm 5 is available in the Appendix).

# 4. EXPERIMENTAL RESULTS

In this section we describe our experiments and discuss their results, focusing on qualifying the impact of algorithms, datasets, and architectures on performance.

## 4.1 Experimental Setup

Due to the portability of OpenCL, we are able to use the same implementations of BFS, APSP, and BC for two different families of architectures: multi-core CPUs and GPUs. The platforms we have used for our experiments are presented in Table 1. For all our CPU experiments, we have used Intels's OpenCL SDK, version

Table 1: The hardware platforms used for the experiments.

| Name | Memory | Bandwidth | PUs |
|---|---|---|---|
| CPU: Intel Xeon E5620 | 24 GB | 25.6 GB/s | $8 \times 2$ |
| GPU: Tesla C2050 | 3 GB | 144 GB/s | $14 \times 32$ |
| GPU: GeForce GTX480 | 1.5 GB | 177.4 GB/s | $15 \times 32$ |

"PUs" stands for processing units (CPU hardware threads or GPU threads). It is computed as the number of "cores" (CPU cores or GPU multiprocessors) $\times$ the number of threads per core.

2.0. For the GPU experiments, we have the OpenCL implemntation from NVIDIA CUDA 5.0.

The results presented in this work focus on the performance of the kernels of each of these algorithms. We note that the overhead due to the data transfer between the host (CPU) and the device (GPU) is not included. This overhead is only significant for BFS (due to its low-processing nature). Indeed, for BFS, when including the data transfer overhead, more datasets show better overall peformance for the CPU instead of the GPU (namely, 1M, ES, 64K, WV, and 4K). However, taking this overhead into account does not change *the variability* of the results which, as seen below, is the most important observation in all our experiments. Moreover, we believe that in realistic scenarios, BFS, APSP, and BC are not computed in isolation, but rather used in more complex pipelines and/or iterative applications. In these cases, the one-time overhead of data copying can be ignored.

## 4.2 Datasets

As datasets, we chose eleven graphs covering a large spectrum of variants in terms of numbers vertices and edges, diameters, and in/out edge ratios. Our datasets are of three types: (i) synthetic graphs, i.e. randomly generated graphs with predetermined properties, (ii) real world graphs, i.e. subsets of existing networks like road networks, social media networks, and email exchange networks, and (iii) pathological graphs, i.e. artificial graphs reflecting the expected worst and best case performance of the BFS, namely a chain of vertices and a star. The three synthetic datasets we use are from the Rodinia benchmark [6], and the remaining six real world datasets are from the SNAP repository [20]. The properties of all our 11 datasets are presented in Table 2.

## 4.3 BFS results

We execute our OpenCL implementation on the three different hardware platforms and measure its execution time for all the 11 datasets and present our results in Figure 1[5]. Note that the results are normalized to the slowest platform.

We point out that he slowest and the fastest platforms are not always the same, showing high variability with the dataset and the platform. We further make the following observations. First, the performance difference between the CPU and the GPUs is similar: either the CPU outperforms both GPUs, or the other way around. Furthermore, GTX always shows a higher performance than Tesla, which is due to the larger bandwidth of the former, an important advantage for memory-bound kernels such as BFS. For large graphs, the performance gap can be as large as 20%, indicating that Second, different graphs show very different preferences for platforms. For example, for WT, the GPUs significantly outperforms the CPU. CR and SW, on the other hand, perform best on the CPU. Three, and last, the sizes of the graphs do not show any immediate correlation

[5]A full overview of all the results is beyond the scope of this paper - therefore, more details can be seen in [25], Chapter 3

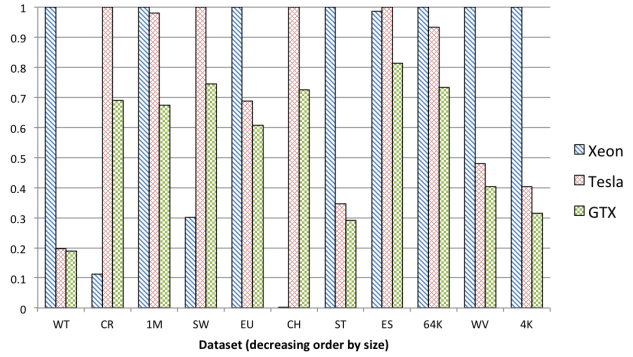



Figure 1: Parallel BFS performance for the Xeon CPU, the Tesla GPU, and the GTX GPU. The execution time is normalized compared with the slowest platform, which shows a relative performance of 1.

with the execution time. For example, the execution time difference between WT and CR is an order of magnitude, despite a mere 20% difference in size. This gap is probably explained by the BFS traversing the node with the maximum numbers of connections (over 100,000 - see Table 2).

Overall, BFS is very sensitive to low parallelism in the input dataset: graphs with low connectivity between nodes are not be able to fully exploit parallel platforms, and their performance is not significantly improved when compared with reference sequential implementations. This behavior is illustrated best by the Chain and Star graphs, which show extreme low and high performance, respectively.

To verify whether we introduced this chaotic behavior through our edge-based implementation, we compare the performance of our edge-based BFS with that of a vertex-based BFS, also implemented in OpenCL, and included in the Rodinia benchmark suite [6]. The results, presented in Figure 2, show large performance discrepancies between the two implementations, with no clear winner. This in turn means that not only is performance significantly dependent on the structure of the dataset, but it is also very sensitive to implementation choices such as graph representation.

In summary, BFS and its parallelism are dominated by the amount of parallelism available in the dataset. Furthermore, being a memory-bound application, using platforms with higher bandwidth can bring additional performance. Finally, using a platform that exceeds the parallelism of the graph (i.e., its average or max connectivity) will lead to severe platform underutilization and, consequently, the gap between the achieved and expected performance will increase.

## 4.4 APSP results

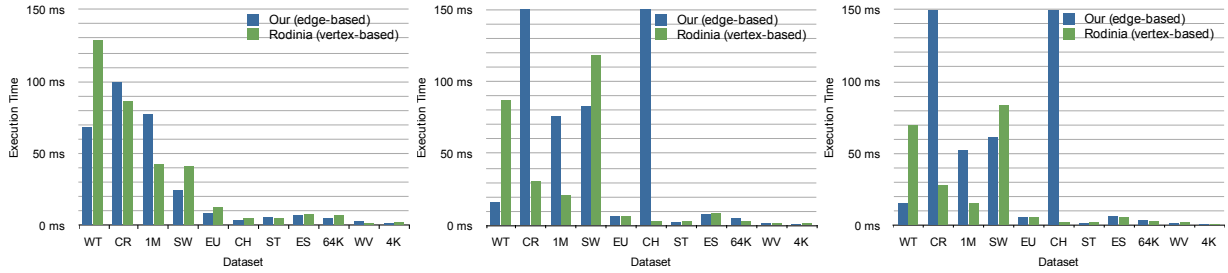
Figure 3 shows the performance results for our APSP implementation in OpenCL.

We make the following observations. First, the GPUs clearly outperform the CPU for all datasets. The achieved speedups are between 1.4 and 11.4 for GTX, and between 1.2 and 6.8 for Tesla. This gain is a direct consequence of the increased parallelism of our APSP algorithm, which matches the massively-parallel GPUs much better than the CPU. Second, we note that the two GPUs perform similarly, with an advantage for the GTX. This is not surprising, given that our APSP is, in fact, a massively concurrent execution of the BFS, which also shows better performance for the GTX card.

In summary, these results reflect the fact that our choice for im-

Table 2: The datasets used in our experiments. *D*, *AVG* and *MAX* represent the diamter, the average and maximum number of vertex connections, respectively.

| Graph name | Vertices | Edges | D | AVG | MAX | Source |
|---|---|---|---|---|---|---|
| Wikipedia Talk Network (WT) | 2,394,385 | 5,021,410 | 9 | 4.19 | 100,032 | SNAP |
| California Road Network (CR) | 1,965,206 | 5,533,214 | 850 | 5.63 | 24 | SNAP |
| Random 1M (1M) | 1,000,000 | 5,999,970 | 11 | 12.00 | 36 | Rodinia |
| Stanford Web Graph (SW) | 281,903 | 2,312,497 | 740 | 16.41 | 38,626 | SNAP |
| EU Email Communication (EU) | 265,214 | 420,045 | 13 | 3.17 | 7,636 | SNAP |
| Chain 100K (CH) | 100,000 | 99,999 | 99,999 | 2.00 | 2 | synthetic |
| Star 100K (ST) | 100,000 | 99,999 | 2 | 2.00 | 99,999 | synthetic |
| Epinions Social Network (ES) | 75,879 | 508,837 | 13 | 13.41 | 3,079 | SNAP |
| Random 64K (64K) | 65,536 | 393,216 | 9 | 12.00 | 48 | Rodinia |
| Wikipedia Vote Network (WV) | 7,115 | 103,689 | 7 | 29.15 | 1,167 | SNAP |
| Random 4K (4K) | 4,096 | 24,576 | 7 | 12.00 | 38 | Rodinia |




(a) Intel Xeon E5620 (b) Nvidia Tesla C2050 (c) Nvidia GeForce GTX480

Figure 2: Comparison of the performance of BFS, for an edge-based versus a vertex-based graph representation: our implementation, and the OpenCL implementation from Rodinia, respectively.

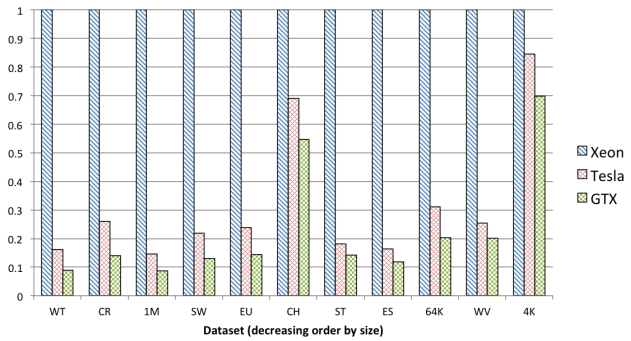



Figure 3: Parallel APSP performance for the Xeon CPU, the Tesla GPU, and the GTX GPU.The results are normalized to the slowest platform, which shows a relative performance of 1.

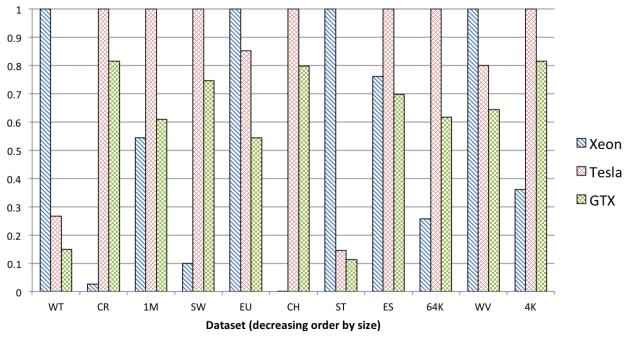



Figure 4: Parallel BC performance for the Xeon CPU, the Tesla GPU, and the GTX GPU. The results are normalized to the slowest platform, which shows a relative performance of 1.

plementing APSP exposes a lot more parallelism than BFS, This increase is directly visible in the kernel performance: the GPUs outperform the CPUs in all cases. For this APSP implementation, "more parallel" architectures handle larger graphs better, as the performance of the kernel is dominated by the number of iterations needed to calculate the whole APSP.

## 4.5 BC results

The performance comparison for computing betweenness centrality on our three platforms is presented in Figure 4.

As the core algorithm of our betweenness centrality is the APSP calculation, we expected the GPUs to outperform the CPU. Instead, we see again different platforms performing best for different workloads.

To clarify this apparent contradiction, we recall that our BC implementation first does a BFS from a vertex $v$, determining the depth of every other vertex. Then it performs a bottom-up traversal from every vertex back to the root $v$, annotating every vertex with path information. This bottom-up traversal takes the same number of iterations as the initial BFS. After this the algorithm iterates over every vertex, updating their betweenness score. Like APSP, this process is done for every vertex. This is why we expected, naïvely, to see a graph similar to that of APSP, given that BC effectively does twice the number of traversals of APSP plus another $|V|$ traversals of all vertices. However, the results shown in Fig-

ure 4 do not confirm this expectation. This behavior comes from the fact that, unlike APSP, BC has to update the state of a vertex multiple times during the traversals. These updates are done using atomic operations to avoid race conditions. Doing many parallel traversals can result in high contention at these atomic operations, reducing the performance. Because the atomic operations are much more expensive on GPUs than on CPUs, the performance penalty is much more significant, relative to the rest of the computation, for the GTX and Tesla.

To verify this intuition we replaced the atomic operations with non-atomic ones - this produces incorrect results, but eliminates the contention. Indeed, when we ran the algorithm on our datasets graph, we saw a significant reduction in runtime for the GPU platforms, confirming that the contention due to atomic operations is having a significant impact on the performance of BC on the GPUs.

## 4.6 Performance per platform

We revisit our experimental results, and present them clustered per platform. Thus, Figure 5 presents the results for the Xeon CPU and the GTX GPU. Presented this way, our results demonstrate the impact of incremental complexity on the two different families of platforms[6].

We make the following observations. First, for the CPU, the increase in complexity leads to an increase in execution time. Not surprisingly, the APSP takes a lot longer than the BFS, given the scale of the graphs and the limited number of available cores (16). Futhremore, BC takes twice longer than APSP, due to the double reuse of APSP inside the BC algorithm. Note that the differences in performance are not the same between data sets, since they differ in sizes and structure.

Second, for Nvidia GeForce GTX480, the performance differences between the three algorithms vary a lot. For example, for the *CR* data set, the fraction of the increase in the execution time between BFS and APSP is similar to that between APSP and BC. However, for the *WT* data set, the difference between BFS and APSP is much larger than that between APSP and BC. These results show that the GPUs (again, we see the same behavior for NVIDIA Tesla) are much more sensitive to the structure of the data set. We note that the GPU performance gap between the APSP and BC is in general very large. We believe this is an effect of the data dependencies for BC calculations: the ratios of each of the shortest paths needs to be derived and accumulated for all vertices (i.e., each ratio depend on the ratios of the neighboring vertices).

Third, we note the different behaviors between the CPUs and GPUs: the differences in dataset size and structure affect the platforms *in different ways*. We see this as additional proof for considering a graph processing workload as an (algorithm, dataset) pair, and potentially choose a matching target using the characteristics of the pair, and not of the algorithm in isolation.

## 4.7 Performance in Meps

A "traditional" measure for high-performance graph processing is *edges per second (eps)* - i.e., how many edges are traversed by the graph processing application in one unit of time (1s). *eps* offers a normalized view of performance, as it implicitly takes into account the size (through normalization) and the structure of the graph (which impacts the execution time) [7].

An accurate measurement of the number of edges traversed in total would require the addition of several counters inside the algorithms, which might in turn change the algorithm behavior. Therefore, we choose to estimate the EPS for each of the algorithms by using the theoretical number of edges they would traverse (i.e., based on the algorithm itself).

For BFS, multiply the diameter $D$ of the graph (which approximates the number of iterations) with the number of edges $M = |E|$ (which approximates the number of edges visited in each iteration - i.e., all of them) and divide by the execution time of the BFS ($T_{BFS}$):

$$EPS_{BFS} = \frac{D \cdot M}{T_{BFS}} \tag{3}$$

For APSP, we use multiple BFS searches, hence we can use a similar approach for computing the EPS: we multiply the diameter of the graph ($D$) with the number of edges ($M$) and with the number of shortest path searches ($N = |V|$). This number of edges traversed is then divided by the execution time of the APSP ($T_{APSP}$):

$$EPS_{APSP} = \frac{D \cdot M \cdot N}{T_{APSP}} \tag{4}$$

For BC, the metric is more difficult to calculate (i.e. it uses an APSP and additional computations to derive the centrality values). In section 4.5, we see that the computations for the centrality values require backtracking of the shortest paths, making it similar to traversing the search tree, in terms of traversal steps. Hence, for simplicity, we can reduce this to an APSP to find the shortest paths in the graph and an APSP to backtrack these shortest path in deriving the centrality values. This simplification allows us to derive a formula for the eps of BC: we multiply the diameter of the graph ($D$) with the number of edges ($M$) and with the number of shortest path searches ($N$), this is divided by the execution time of the BC ($T_{BC}$). This value is then multiplied by two, resulting in:

$$EPS_{BC} = 2 \cdot \frac{D \cdot M \cdot N}{T_{BC}} \tag{5}$$

Figure 6 presents the performance of BFS, APSP, and BC in *Meps*.

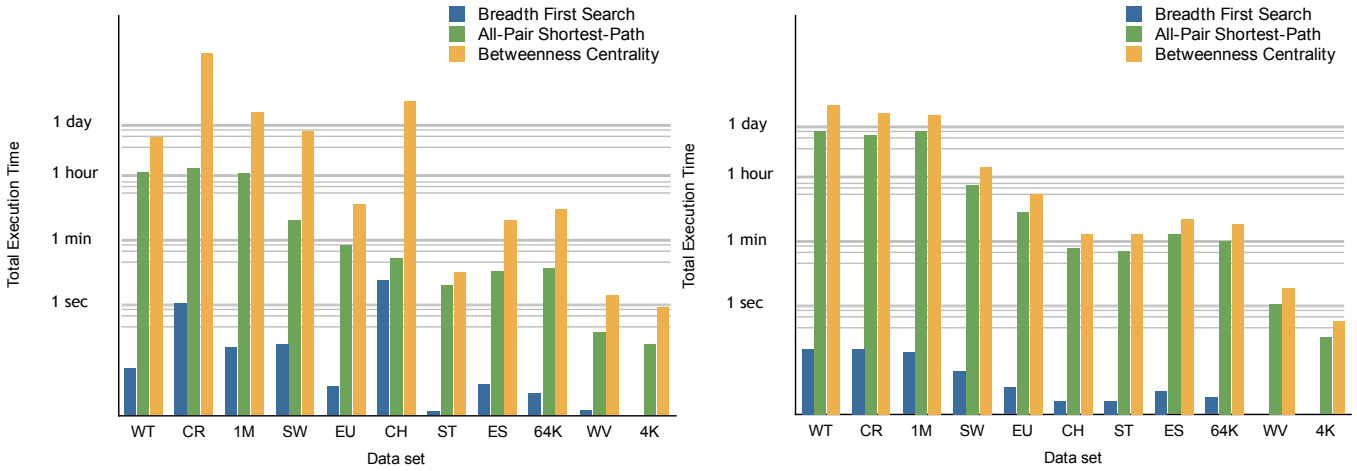
We make the following observations: First, APSP shows the best *eps*, regardless of the platform or data set. This is because the algorithm we have chosen for APSP is a massively parallel one, and, in combination with the edge-based representation of the graph, it is suitable for the chosen parallel platforms. Second, for BC, the number of edges traversed/processed per second is lower, due to the additional complexity of the computation performed to determine the BC coefficients.

Third, we note again the different behavior of the two hardware platforms. For the GPUs, a large performance gap is visible between APSP and the rest, caused by the large potential of parallelism of our algorithm. For the CPU, the gap is smaller due to coarser level of parallelism in the system. The only graph that breaks the pattern is *ST*, which seems to favor fine-grained parallelism also for the BC algorithm.

Overall, although *eps* provides normalized performance against the graph size, it remains insufficient for quantifying/qualifying the impact of the dataset structure on the overall performance of the algorithm. In other words, *eps* can be used to compare different implementations of the same algorithm, but provides too little insight into matching workloads to platforms.

[6]We only show the CPU and GTX 480 behavior due to because the Tesla performance trends are very similar to those of the GTX.

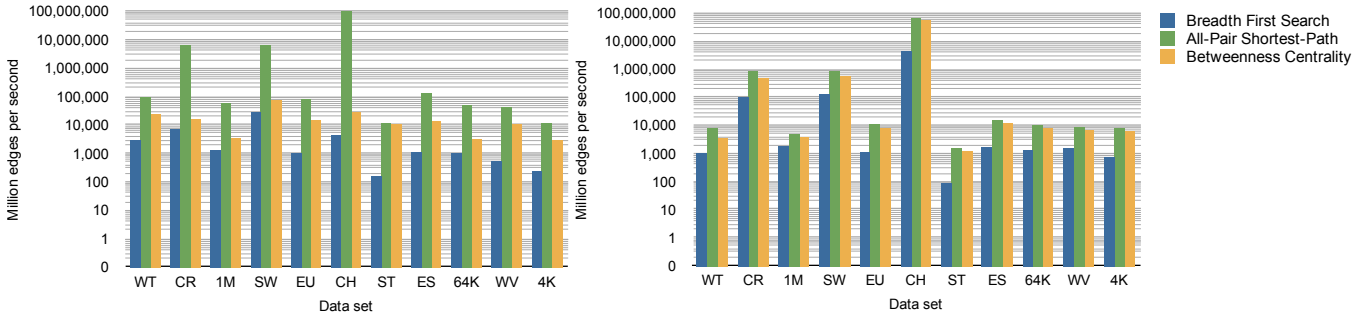
[7]This is also the metric used in Graph500.




(a) NVIDIA GTX480

(b) Intel Xeon E5620

Figure 5: The performance of our BFS, APSP, and BC (logarithmic scale) for the GTX and Xeon platforms.




(a) NVIDIA GTX480

(b) Intel Xeon E5620

Figure 6: Million edges per second for the BFS, APSP, and BC algorithms, grouped per dataset.(logarithmic scale).

## 5. RELATED WORK

We discuss in this section two different related work fields: modern studies of parallel graph processing, targeted at multi- and many-core architectures, and studies on performance portability.

Work on parallel graph processing stands out by its specific approaches: improved parallel algorithms, tuning parallel algorithms to effectively use of modern parallel hardware, or building high-level programming solutions to hide parallelism from programmers.

For example, previous work improves the throughput of graph searches (based on BFS) on parallel systems by data partitioning and message compression [33]. The queuing of visited graphs vertices (also seen in BFS) have been replaced in [19] with *bags*, allowing splitting and merging of queued operations such that they can operate in parallel. For a centrality algorithm, recent work [21] replaces the SSSP algorithm, used to calculate the pair-dependencies for all pairs, with a BFS algorithm (as suggested in other work [4]), and gain additional performance. Finally, work presented in [7, 27] shows how exploiting the lowest-level specifics of hardware archiecture can lead to a high performance algorithm for graph processing, which will outperform most of the competition, yet it will not work efficiently on any other machine than the one it is designed for.

Using GPUs to accelerate graph processing has also been tried, by adapting traditional graph analysis to heterogeneous systems [8]. In [14, 31], the quantify the performance gain of GPU-enabled graph analysis against the sequential version. The GPU shows significant performance improvement for different algorithms, but dataset impact is not quantified. In [17, 29], graph algorithms are adapted for GPU execution by reordering operations and data accesses.

All these examples show that the research on modern architectures for graph analysis focuses on *specific algorithms*, and their improvement for a *specific environment*. In contrast to our work, none of these studies quantifies the impact of the datasets on the achieved performance, nor does it address the portability of the approaches to a new/different generation of parallel architectures. While the latter is, for many of them, out of the scope of their research, the former is mandatory for a better understanding of the observed performance.

An alternative approach to using CPUs and GPUs for graph processing has emerged in the form of dedicated programming systems. MEDUSA [37] and TOTEM [12] are two examples of systems take into account some of the properties of the datasets to schedule, at runtime, the computation of a graph analysis algorithm on the different components of a heterogeneous system. TOTEM is further improved in [13] with a systematic CPU-GPU partitioning algorithm that takes into account, at runtime, the characteristics of the dataset and the hardware. Ultimately, we share the goal of matching datasets with architectures, but this scheduling-driven approach in [13] does not pay enough attention to the algorithm; we believe an analytical model that mixes dataset, algorithm, and platform will be more useful in the context of portability.

The second dimension of our work is related to performance and portability (in the context of OpenCL), a controversial concept in the current landscape of parallel processing. Several empirical studies on performance portability [28, 18, 34] have shown, for different mixes of applications and platforms (all including GPUs and CPUs), that OpenCL provides a good basis for performance portability. We note, however, that (1) none of the used benchmarks were irregular applications, and (2) no dependence of the performance on the dataset has been studied. Similar approaches in [36, 30] have taken a more proactive approach, and showed how OpenCL applications can be improved in terms of performance portability. Again, no discussion on irregular algorithms or dataset performance dependency is included.

Another way to achieve portability is to build platform-agnostic algorithms. In [26], the authors introduced the notion of *amorphous data parallelism*, an attempt to parallelize highly irregular algorithms by exposing the fine-grained parallelism within the algorithm, which in turn can scale better with the available hardware parallelism. Several studies on using amorphous data parallelism on GPUs have looked at the performance of graph algorithms [5, 24]. However, these studies take the use of a GPU as a given, and do not compare the GPU performance against CPU performance. As our results demonstrate several cases when CPUs outperform GPUs, such a comparison is necessary to prove the portability and the high performance of these solutions.

This brief survey of related work demonstrates that performance portability and parallel graph processing have been, so far, disjoint. In this context, our work is the first to combine them in claiming that performance portability assessment for irregular applications must take all three variables - dataset, application, and platform - into account. We finally argue (claim indirectly supported by the work in [12, 13]) that portability allows for better usage of heterogeneous platforms, contributing to an overall increase in performance, especially in the context of large scale datasets.

# 6. CONCLUSION

Graph processing is one of the pillars of big data analysis, and it becomes a real challenge for high performance computing and its multi-layer massively parallel architectures of today (multi-core CPUs and GPUs). Many different parallel graph analytics algorithms have been proposed, all addressing the specific needs of one hardware platform or another, and achieving good performance for various datasets *on their target platform*. However, these solutions are rarely evaluated, in terms of performance gain/loss in comparison with different parallel platforms [22] and using a truly wide variety of graphs. Without such an evaluation, users' have little flexibility in their choices: platform comes first.

We argue that a landscape with so many different flavors of graph analytics algorithms and such incomplete evalution, is difficult to navigate by regular users that want to focus more on productivity and efficiency than on software development and platform updates. For such users, our work has showed that using relatively simple, portable graph algorithms, one can make use of most of their in-house resources without code changes. Due to the strong dependency of graph analytics (such as BFS, APSP, and BC) on the structure of the datasets, the performance loss *across different types of graphs is not significant* - i.e., depending on the algorithm and dataset, the right platform to be used can be a GPU, a CPU, or a combination of the two. By having the same implementation for both types of platforms (and many others, given the portability of OpenCL), users can easily maintain a single codebase, and perform a simple empirical check to find the right platform for a new dataset. In this sense, portability can offer a performance boost by providing multiple platform options: for algorithms such as BFS and BC, over 40% of our tests show a CPU outperforming the GPUs. We acknowledge that this approach might not offer the best ever performance for a (platform,dataset,algorithm) at hand, but we argue none of the alternatives will. The only option for finding the absolute best remains a trial-and-error sweep to rank *all* the existing solutions, a prohibitively expensive, low-productivity approach given the tens of alternatives.

To summarize, our results have shown that graph analytics portability (using OpenCL) is feasible for CPUs and GPUs. Our portable algorithms provide additional performance opportunities by simply allowing alternative execution platforms to be used. We observed that such flexibility pays off: we gained significant performance in more than 40% of the cases we tested, by simply replacing a GPU with a CPU.

Our future work will focus on two directions: performance improvement and analytical modeling. On the short term, we aim to improve the performance of these portable implementations by generalizing the findings of the most promising platform-specific solutions (see Section 5). On the longer term, we aim to build models of impact of both platforms and datasets characteristics on the performance of the graph analytics workloads. Using such models, we aim to predict the best platform for a given (dataset,algorithm) pair for portable, platform-agnostic algorithms.

# References


[1] A. LUMSDAINE, D. GREGOR, B. H., AND BERRY, J. W. Challenges in parallel graph processing. *Parallel Processing Letters 17* (2007).

[2] AGARWAL, V., PETRINI, F., PASETTO, D., AND BADER, D. A. Scalable graph exploration on multicore processors. In *SC* (2010), pp. 1–11.

[3] BELLMAN, R. On a routing problem. *Quarterly of Applied Mathematics, vol. 16* (1958), 87–90.

[4] BRANDES, U. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology 25* (2001), 163–177.

[5] BURTSCHER, M., NASRE, R., AND PINGALI, K. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on* (2012), IEEE, pp. 141–151.

[6] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S. H., AND SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. *The 2009 IEEE International Symposium on Workload Characterization, IISWC'09* (2009), 44–54.

[7] CHECCONI, F., AND PETRINI, F. Massive data analytics: The graph 500 on ibm blue gene/q. *IBM Journal of Research and Development 57*, 1/2 (2013), 10.

[8] DINNEEN, M. J., KHOSRAVANI, M., AND PROBERT, A. Using opencl for implementating simple parallel graph algorithms. *Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'11* (2011).

[9] FLOYD, R. W. Algorithm 97: Shortest path. *Communications of the ACM* (1962), 345.

[10] FORD, L. R., AND FULKERSON, D. R. Flows in networks. *Princeton University Press* (1962).

[11] FREEMAN, L. C. Centrality in social networks conceptual clarification. *Social Networks* (1978), 215.

[12] GHARAIBEH, A., COSTA, L. B., SANTOS-NETO, E., AND RIPEANU, M. A yoke of oxen and a thousand chickens for heavy lifting graph processing. *The 21st international conference on Parallel architectures and compilation techniques, PACT'12* (2012), 345–354.

[13] GHARAIBEH, A., COSTA, L. B., SANTOS-NETO, E., AND RIPEANU, M. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *IPDPS* (2013), pp. 851–862.

[14] HARISH, P., AND NARAYANAN, P. J. Accelerating large graph algorithms on the gpu using cuda. *HiPC'07 Proceedings of the 14th international conference* (2007).

[15] J. FANG, A. L. V., AND SIPS, H. A comprehensive performance comparison of cuda and opencl. *The 40th International Conference on Parallel Processing, ICPP'11* (2011).

[16] JOHNSON, D. B. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM 24 issue 1* (1977).

[17] KATZ, G. J., AND JR, J. T. K. All-pairs shortest-paths for large graphs on the gpu. *23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2008), 47–55.

[18] KOMATSU, K., SATO, K., ARAI, Y., KOYAMA, K., TAKIZAWA, H., AND KOBAYASHI, H. Evaluating performance and portability of opencl programs. In *The Fifth International Workshop on Automatic Performance Tuning* (June 2010).

[19] LEISERSON, C. E., AND SCHARDL, T. B. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). *Computer Vision and Pattern Recognition, CVPR'10* (2010), 2181–2188.

[20] LESKOVEC, J. Stanford network analysis platform (snap). *Stanford University* (2006).

[21] MADDURI, K., EDIGER, D., JIANG, K., BADER, D. A., AND CHAVARRIA-MIRANDA, D. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. *The 2009 IEEE International Symposium on Parallel and Distributed Processing, IPDPS'09* (2009), 1–8.

[22] MERIJN VERSTRAATEN, ANA LUCIA VARBANESCU, C. D. L. State-of-the-art in graph traversals on modern arhictectures. Tech. rep., University of Amsterdam, August 2014.

[23] N. EDMONDS, A. BREUER, D. G., AND LUMSDAINE, A. Single-source shortest paths with the parallel boost graph library. *The Ninth Implementation Challenge: The Shortest Path Problem* (2006).

[24] NASRE, R., BURTSCHER, M., AND PINGALI, K. Data-driven versus topology-driven irregular computations on gpus. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* (2013), IEEE, pp. 463–474.

[25] PENDERS, A. Accelerating Graph Analysis with Heterogeneous Systems. Master's thesis, PDS, EWI, TUDelft, December 2012.

[26] PINGALI, K., NGUYEN, D., KULKARNI, M., BURTSCHER, M., HASSAAN, M. A., KALEEM, R., LEE, T.-H., LENHARTH, A., MANEVICH, R., MÉNDEZ-LOJO, M., ET AL. The tao of parallelism in algorithms. *ACM SIGPLAN Notices 46*, 6 (2011), 12–25.

[27] QUE, X., CHECCONI, F., AND PETRINI, F. Performance analysis of graph algorithms on p7ih. In *ISC* (2014), pp. 109–123.

[28] RUL, S., VANDIERENDONCK, H., D'HAENE, J., AND DE BOSSCHERE, K. An experimental study on performance portability of opencl kernels. In *Application Accelerators in High Performance Computing, 2010 Symposium, Papers* (2010), p. 3.

[29] S. HONG, S. K. KIM, T. O., AND OLUKOTUN, K. Accelerating cuda graph algorithms at maximum warp. *Principles and Practice of Parallel Programming, PPoPP'11* (2011).

[30] SHEN, J., FANG, J., SIPS, H., AND VARBANESCU, A. Performance gaps between openmp and opencl for multi-core cpus. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on* (Sept 2012), pp. 116–125.

[31] SRIRAM, A., AND GAUTHAM, K. Evaluating centrality metrics in real-world networks on gpu. *High Performance Computing, HiPC'09 Student Research Symposium 2009* (2009).

[32] U. MEYER, V. O. Design and implementation of a practical i/o-efficient shortest paths algorithm. *10th Workshop on Algorithm Engineering and Experiments, ALENEX'09* (2009), 85–96.

[33] UENO, K., AND SUZUMURA, T. Highly scalable graph search for the graph500 benchmark. *The 21st international symposium on High-Performance Parallel and Distributed Computing, HPDC'12* (2012), 149–160.

[34] VAN DER SANDEN, J. Evaluating the performance and portability of opencl. Master's thesis, TU Eindhoven, 2011.

[35] VARBANESCU, A. L., AND IOSUP, A. On many-task big data processing: From GPUs to clouds.

[36] YAO ZHANG, M. S. I., AND CHIEN, A. A. Improving performance portability in opencl programs. In *International Supercomputing Conference 2013* (2013).

[37] ZHONG, J., AND HE, B. Medusa: Simplified graph processing on gpus. *17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP'12* (2012), 283–284.

# APPENDIX

## A. THE BFS, APSP, AND BC ALGORITHMS

In this section we include all the pseudocode samples that illustrate our sequential and parallel (i.e., OpenCL) implementations of BFS, APSP, and BC.
Our sequential BFS is presented in Algorithm 1.

**Algorithm 1** Edge-based BFS implementation for graph $G = (V, E)$, using start vertex $s$; $N = |V|$ is the number of vertices and $M = |E|$ is the number of edges.

```
1: function EndgeBased_BFS(E, s)
2:     Q ← ∅
3:     for e ∈ E do
4:         e ← UNVISITED
5:     end for
6:     add(Q, s)
7:     changed ← 1
8:     while changed = 1 do
9:         changed ← 0
10:        for e ∈ E, e = UNVISITED do
11:            if Source(e) ∈ Q then
12:                add(Q, Dest(e))
13:                e ← VISITED
14:                depth[Dest(e)] ← depth[Source(e)] + 1
15:                changed ← 1
16:            end if
17:        end for
18:    end while
19: end function
```

Our sequential APSP is presented in Algorithm 2.

**Algorithm 2** A BFS-based APSP implementation. $R$ is the result as a collection of distances between pairs of vertices.

```
1: function APSP_Using_BFS(V)
2:     R ← ∅
3:     for v ∈ V do
4:         BFS(V, v)
5:         for u ∈ V do
6:             if v ≠ u then
7:                 Add(R, Distance(v, u))
8:             end if
9:         end for
10:    end for
11: end function
```

Our parallel BFS is presented in Algorithm 3.
Our parallel APSP is presented in Algorithm 4.
Our parallel BC algorithm is presented in Algorithm 5.

**Algorithm 3** Parallel BFS implementation in OpenCL, for graph $G = (V, E)$ and start vertex $s$; $E_{thread} \subseteq E$ is the edge list assigned to this thread (PU); operations preceded by $atomic$ are executed atomically; $barrier()$ is a global barrier.

```
1: function KERNEL_BFS(E, s)
2:     Q_i ← {s}
3:     hasNextLevel ← 1
4:     i ← 0                                   ▷ Set Current Level
5:     while hasNextLevel = 1 do
6:         for e ∈ E_thread, e = UNVISITED do
7:             if Source(e) ∈ Q_i then
8:                 ▷ Attempt to lock.
9:                 while (atomic(CAS(Dest(e), 0, 1)))
10:                ▷ Lock succeeded.
11:                add(Q_{i+1}, Dest(e))
12:                e ← VISITED
13:                depth[Dest(e)] ← depth[Source(e)] + 1
14:                ▷ Unlock.
15:                atomic(Dest(e) ← 0)
16:                atomic(hasNextLevel ← 1)
17:            end if
18:        end for
19:        i ← i + 1                           ▷ Increase Level
20:        barrier()                           ▷ Level Synchronization
21:    end while
22: end function
```

**Algorithm 4** Our parallel APSP implementation in OpenCL, for graph $G = (V, E)$, with $R$ being the result distance matrix, $V_group \subseteq V$ being the sources list assigned to this work-group, and $E_{thread} \subseteq E$ being the edge list assigned to this work-item.

```
1: function KERNEL_APSP(E, V)
2:     for s ∈ V_group do
3:         Q_i ← {s}
4:         hasNextLevel ← 1
5:         i ← 0                               ▷ Set Current Level
6:         while hasNextLevel = 1 do           ▷ Begin BFS
7:             for e ∈ E_thread do
8:                 if e = UNVISITED & Source(e) ∈ Q_i
   then
9:                     add(Q_{i+1}, Dest(e))
10:                    e ← VISITED
11:                    R[s][Dest(e)] ← depth[Source(e)] + 1
12:                    atomic(hasNextLevel ← 1)
13:                end if
14:            end for
15:            i ← i + 1                       ▷ Increase Level
16:            barrier()                       ▷ Level Synchronization
17:        end while                           ▷ End BFS
18:        barrier()
19:    end for
20: end function
```

**Algorithm 5** Our parallel BC implementation in OpenCL, for graph $G = (V, E)$, with $BC$ be the resulting vector containing the betweenness centrality scores for all the vertices in $V$.

```
1: function KERNEL_BC(E, V)
2:     for s ∈ V_group do
3:         Q_i ← {s}
4:         σ ← ∅                                                    ▷ Path Count
5:         δ ← ∅                                                    ▷ Pair-Dependency
6:         hasNextLevel ← 1
7:         i ← 0                                                    ▷ Set Current Level
8:         while hasNextLevel = 1 do                                ▷ Step II
9:             for e ∈ E_thread do
10:                if e = UNVISITED & Source(e) ∈ Q_i then
11:                    add(Q_{i+1}, Dest(e))
12:                    e ← VISITED
13:                    σ[Dest(e)] ← σ[Dest(e)] + σ[Source(e)]
14:                    atom_xchg(hasNextLevel, 1)
15:                end if
16:            end for
17:            i ← i + 1                                            ▷ Increase Level
18:            barrier()                                            ▷ Level Synchronization
19:        end while
20:        i ← i − 2
21:        while i > 1 do                                           ▷ Step III
22:            for e ∈ E_thread do
23:                if Source(e) ∈ Q_i & Dest(e) ∈ Q_{i+1} then
24:                    delta ← (σ[Source(e)] / σ[Dest(e)]) · (δ[Dest(e)] + 1)
25:                    atom_add(δ[Source(e)], delta)
26:                end if
27:            end for
28:            i ← i − 1
29:            barrier()                                            ▷ Level Synchronization
30:        end while
31:        for v ∈ V do                                             ▷ Step IV
32:            if v ≠ r then
33:                atomic(BC[v] ← BC[v] + δ[v])
34:            end if
35:        end for
36:    end for
37: end function
```