Breadth-First Search on Heterogeneous Platforms: A Case of Study on Social Networks

Luis Remis*‡, Maria Jesus Garzaran*‡, Rafael Asenjo† and Angeles Navarro†
*Department of Computer Science, University of Illinois at Urbana-Champaign
†Universidad de Málaga, Spain {asenjo, angeles}@ac.uma.es
‡Intel Corp. {luis.remis, maria.garzaran}@intel.com

Abstract—**Breadth-First Search (BFS) is the core of many graph analysis algorithms and it is used in many problems, such as social network, computer network analysis, and data organization. BFS is an iterative algorithm that due to its irregular behavior is quite challenging to parallelize. Several approaches implement efficient algorithms for BFS for multicore architectures and for Graphics Processors, but it is still an open problem how to distribute the work among the main cores and the accelerators. In this paper, we assess several approaches to perform BFS on different heterogenous architectures (high-end and embedded mobile processors composed of a multi-core CPU and an integrated GPU) with a focus on social network graphs. In particular, we propose two heterogenous approaches to exploit both devices. The first one, called Selective, selects on which device to execute each iteration. It is based on a previous approach, but we have adapted it to take advantage of the features of social network graphs (fewer iterations but more unbalanced). The second approach, referred as Concurrent, allows the execution of specific iterations concurrently in both devices. Our heterogenous implementations can be up to 1.56x faster and 1.32x more energy efficient with respect to the best of only-CPU or only-GPU baselines. We have also found that for a highly memory bound problem like BFS, the CPU-GPU collaborative execution is limited by the shared-memory bus bandwidth.**

I. Introduction

Graphs can be used to represent many practical problems, such as social networks, computer networks, and data organization. Breadth-First Search (BFS) is the core of many graph analysis algorithms used in flow network analysis, shortest-path problem, and in many others graph traversal algorithms.

Parallel implementations of BFS are very challenging due to its irregular behavior, heavy memory access and very low arithmetic intensity. When processing large graphs, it becomes necessary to deal with large data structures and lack of memory locality, since random memory access prevents locality optimizations. The complexity of BFS is $O(N+M)$, where N is the number of vertices and M is the number of edges. However, the execution time depends not only on the number of vertices and edges, but also on its structure, as it will be discussed in the next sections.

The use of GPUs coupled to multicores to speed up applications has become very common, especially on data parallel algorithms with regular memory accesses [1], [2]. The use of GPUs to compute irregular algorithms, such as BFS, is tempting because of GPU's high throughput, although an efficient parallelization is harder.

In a system where a discrete GPU is present, the work flow involves transferring data from main memory to GPU device memory, performing the computations on the GPU, and moving the results back to main memory. This data movement is done through the PCI express bus, which is orders of magnitude slower than making copies within the main memory. Heterogenous platforms with an integrated GPU, on the other hand, offer an environment where memory is shared between the CPU and the accelerator (in this case, OpenCL capable GPU). Thus, all the data movement can be avoided. The OpenCL specification offers the programmer the necessary APIs to allocate shared memory blocks that can be addressed from both devices [3].

The goal of this work is to explore how to efficiently implement BFS on heterogenous systems, analyzing different strategies for distributing the work among the CPU and the GPU devices. In particular, we make the following contributions. First, two heterogeneous approaches have been implemented: one that dynamically selects the best device to run (CPU or GPU), and another that uses both devices concurrently (Sec. III). The relevant details of the implementation are summarized (Sec. IV), relying on C++, OpenMP, and OpenCL to produce a code portable for the different platforms we used. Next, our work distribution strategies are assessed with 10 social network graphs and 3 road network graphs on three different heterogenous platforms (Sec. V). When using Only-CPU (including state-of-the-art Galois [4]) and Only-GPU implementations as baseline, our heterogeneous implementations obtain up to 1.56x speedup and 1.32x energy efficiency. However, memory congestion is a key bottleneck when both devices run simultaneously, which represents still an open problem for future work.

II. Background on BFS

Given an undirected graph $G(V,E)$ and a source vertex s_0, a breadth-first search traverses G starting from s_0 and exploring all the vertices at a given distance d from s_0 before traversing further vertices. The classic Top-Down implementation of BFS is shown in Algorithm 1, which results in an array of distances for every vertex with respect to s_0.

A vertex frontier is the subset of vertices that are discovered for the first time and are enqueued in each iteration of the serial **while** loop (lines 7-18). The first frontier, F_{old} contains just the source vertex s_0. After each iteration of the parallel **for** (lines 8-17), a new frontier, F_{new} is discovered, which contains all the vertices at distance 1 from the vertices in F_{old}. When all the vertices have been visited, F_{new} becomes empty ($\nexists v \mid dist[v] == \infty$), then $F_{old} = \emptyset$ and the algorithm ends.

Algorithm 1 Breadth-First Search

Input: Graph (V,E), source vertex s_0
Output: Distances from s_0 $Dist[1..|V|]$

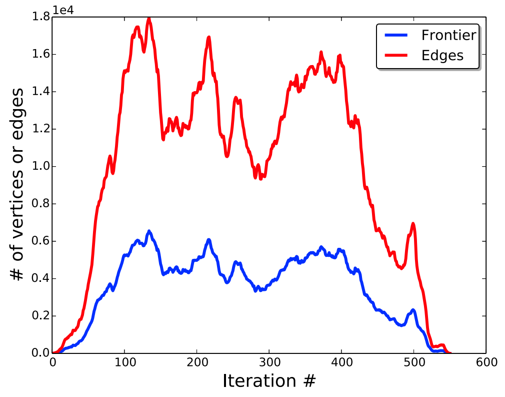
```
1:  for all $v \in V$ do
2:      $dist[v] = \infty$
3:  end for
4:  $dist[s_0] = 0$
5:  $F_{old} = \emptyset$
6:  $F_{old}.enqueue(s_0)$
7:  while $F_{old} \neq \emptyset$ do                          ▷ Serial while
8:      for all $i$ in $F_{old}$ do                            ▷ Parallel loop
9:          for each neighbor $v$ of $i$ do
10:             if $dist[v] == \infty$ then
11:                 $dist[v] = dist[i] + 1$
12:                 $F_{new}.enqueue(v)$
13:             end if
14:         end for
15:         $F_{old} = F_{new}$
16:         $F_{new} = \emptyset$
17:     end for
18: end while
```

Coarse-grained parallelism can be exploited using OpenMP to distribute chunks of the iteration space of the parallel loop among the cores. However, the GPU excels at exploiting fine-grained parallelism, which can be put to work by processing each vertex i of the parallel loop by a GPU thread.

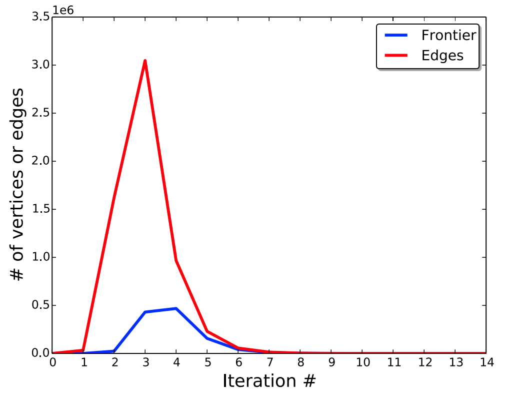
The Top-Down algorithm just described may perform redundant work since some neighbors v of a vertex i may have already been visited in this or in previous iterations of the while loop (these v's are already part of the new frontier F_{new} or of the previous F_{old} frontiers). This incurs in a waste of computing time and power.

An alternative implementation that avoids this problem is known as the Bottom-Up algorithm. With this alternative, the vertices that have not been discovered yet, check if they have a parent in the frontier. To do that, each non-discovered vertex traverses its edges. If a vertex finds a parent in the frontier it updates its distance and adds itself to the new frontier. This way, a vertex that finds a parent in the frontier, stops processing the rest of its edges, eliminating the redundant work. Such situation is likely to occur when an important part of the graph has already been processed. The disadvantage of this method is that all unvisited vertices need to be explored, and for those that do not find a parent in the frontier, the algorithm performs useless work and wastes power.

An extensive exploration about which method (Top-Down /Bottom-Up – coarse/fine grained) is more convenient to use has been done in [5]. The Top-Down method has the advantage that only the vertices in the frontier are explored, whereas the Bottom-Up method needs to explore all the unvisited vertices. Using the Bottom-Up method in the first iterations of the graph traversal will result in the exploration of the majority of the vertices (as only a few would have been visited in the beginning) and in the exploration of almost all the neighbors (as it would be difficult for a vertex to find a parent in the frontier because this will be small). However, using the Top-Down method during the first iterations will result in less redundant work, as only the vertices in the frontier will be explored, and the majority of their neighbors will be unvisited at that point. As iterations execute vertices are visited and there is a point when the Bottom-Up method performs less useless work, whereas the Top-Down performs a larger amount of redundant work.



(a) Roads of California: 1.96M Vertices



(b) Youtube Graph: 1.17M Vertices

Fig. 1. Comparison between Social Network and Road Graphs.

A. Social Network Graphs

BFS applied to social network graphs started to gain relevance recently due to the proliferation of social network services. Figure 1 represents, for all the iterations of the *Serial while* of Algorithm 1, i.e. for each frontier, how many vertices it has, *Frontier* line, and how many edges are traversed when processing it, *Edges* line. Figure 1(a) shows the case of a regular graph like the roadCA graph. This graph represents a street map, where every vertex has a similar and small number of edges. This translates into several iterations (more than 500) needed to reach every vertex in the graph, where all the iterations perform a similar amount of work. As the figure shows, the number of visited vertices and traversed edges are within the same order of magnitude across iterations.

On the other hand, Figure 1(b) depicts the scenario on social networks. In this case, the Youtube social network is traversed in only 14 iterations. The plot also shows that the amount of edges explored by some iterations (2 to 4 in this graph) is one order of magnitude higher than for the rest of the iterations. This presents both an advantage and challenge for heterogenous platforms, as the number of iterations (and thus, synchronization) is small, but the workload is not evenly distributed among the iterations. The need for fewer iterations when traversing the graph brings new opportunities to platforms that are very sensitive to synchronization barriers (implicit or explicit), such as GPUs, and thus, heterogenous platforms may find its way for this kind of graphs. In particular, having fewer iterations means fewer barriers and overheads; in addition, having a higher amount of work per iteration might be profitable for GPU acceleration.

III. Approaches

We have designed and implemented two heterogeneous approaches for performing BFS. In the first approach, referred as `Selective`, some iterations of the *Serial while* of Algorithm 1 (line 7), i.e. some frontiers, are processed on the CPU while others are processed on the GPU. The device is selected based on the size of the frontier and on how many vertices have already been visited. With this approach, both devices are never exploited simultaneously. The second approach is called `Concurrent`, where both devices are used at the same time to process some specific frontiers. These frontiers are split between the devices and the graph is traversed concurrently. We further elaborate on these approaches next.

A. Selective

The `Selective` approach is based on the observation that the CPU can process a given frontier faster than the GPU when the number of vertices in the frontier is small. However, when the frontier is big enough, the GPU can extract enough parallelism and explore the frontier faster than the CPU cores. Therefore, the `Selective` approach starts processing the first frontiers on the CPU, and switch to the GPU when the number of vertices in the frontier is above a threshold. With this approach, a frontier is executed on the device which fits best. A similar methodology has been proposed previously by Munguia et al [6], but from the perspective of a task-based framework using discrete GPUs and by Daga et al [7], using a heuristic similar to the one proposed by Beamer et al [5] to switch from using Top-Down on CPU to use Bottom-Up on GPU. Their methodology requires counting both vertices and edges in the frontier, which can add a significant overhead when processing a frontier on the GPU. Counting edges requires exploring the frontier and adding up all the connections of each vertex, which is an expensive operation.

To avoid counting the number of edges in the frontier, in our heuristic we switch from the Top-Down CPU phase to the Bottom-Up GPU phase when more than 10% of the graph has been visited, which is cheaper to detect. We have experimentally evaluated different thresholds, finding that for our graphs this is the one that produces the best results. In fact, for the big social network graphs that we have used in our experiments, after 10% of the graph is processed, the frontier has more than 40K vertices, which makes the GPU the suitable device to process such a big frontier faster. After this point, our `Selective` approach processes three frontiers on the GPU using Bottom-Up approach. To reduce the overhead of GPU kernel launching, three GPU kernels are dispatched to the OpenCL queue. This is an in-order queue where the three frontiers are executed one after the other. In our experiments, these three frontiers usually process around 80% of the vertices and the size of the remaining frontiers are likely to be again too small to be processed on the GPU. If after processing 3 frontiers on the GPU the amount of processed vertices is smaller than 90% (10% in the CPU phase plus 80% in the GPU one), additional kernels are enqueued to the GPU one by one until this conditions is met. Finally, when less than 10% of the vertices remain, the CPU is back in charge of running the last frontiers using the Bottom-Up algorithm.

B. Concurrent

In this approach, both devices are used at the same time to process the frontiers that are large enough. The idea is to further reduce the execution time by simultaneously exploiting the CPU and the GPU. As in the `Selective` approach, when 10% of the vertices have been processed, we switch to a CPU-GPU phase, in which the workload of a parallel iteration is distributed between the GPU and the CPU cores. Section V-C discuss the experimental results obtained when applying different workload partitions. In this phase, in principle, both Top-Down and Bottom-Up methods can be used. If we rely on the Top-Down method, the frontier is created in the CPU from a global bitmask, indicating which are the vertices belonging to the frontier. Then, a portion of those vertices are assigned to the GPU by creating a new GPU-specific bitmask (this incurs a high overhead). On the other hand, the Bottom-Up approach does not involve any new GPU-specific bitmask creation. As we explain in the next section, the `Concurrent` Bottom-Up approach splits the vertices between the CPU and the GPU. Then, both devices will process their own set of vertices concurrently. For each vertex, v, the CPU or the GPU has to check if it has not been visited yet ($dist[v] == \infty$) and in that case if it has a neighbor, i, in the old frontier, F_{old}. For this, it reads the corresponding flag in the global bitmask. If this is the case, the vertex's distance is updated ($dist[v] = dist[i]+1$) and the vertex is added to the new frontier, F_{new}, updating a different global bitmask (the new frontier bitmask). Since the CPU and the GPU are traversing different vertices, they are reading different regions of the distance vector and of the old frontier bitmask, as well as writing to different non-overlapping regions of the distance vector and new frontier bitmask. Therefore this Bottom-Up alternative can be heterogeneously implemented without introducing any overhead, and it is the chosen approach to process the large frontiers of the graph. As in the `Selective` approach, when more than 90% of the vertices are processed, we switch back to a Bottom-Up CPU-only phase.

IV. Implementation Details

Our CPU BFS implementations (Top-Down and Bottom-Up) are written using C++ and OpenMP. The data structure to store the graph is, as usual, CSR (Compressed Sparse Representation) [8]. This is, we store the graph in two vectors: i) the vector e with M edges, and ii) the vector v that stores N vertices in a $N+1$ array. According to the CSR format, the edges connecting $v[i]$ are stored in v in the indices indicated by $e[v[i]]...e[v[i+1]-1]$. Three additional vectors of length N are also needed: the distance vector $dist$ and two bitmask vectors bm_{old} and bm_{new}. The bitmasks identify the vertices of v that belong to the *old* and *new* frontiers, respectively, as described in Algorithm 1.

Originally, the CPU Top-Down CPU algorithm was implemented using OpenMP *parallel for* in the loop i, going over the whole bitmask vector, bm_{old}, but processing only the vertices $v[i]$ such that $bm[i] == true$. If the frontier is small, this is suboptimal since $bm[i]$ is usually false (since only a few $v[i]$ belong to the frontier). To avoid this overhead, we optimized the implementation following an inspector-executor approach. First, at the beginning of each iteration of the *Serial while*, the bitmask bm_{old} is explored to produce a dense frontier F_{old} (inspector step). Then, this dense frontier is processed in a parallel loop (executor step). Although the inspector step is implemented in parallel relying on a prefix sum, we found out that for small graphs (with less than 1M vertices), the inspector step does not pay off. However, for larger graphs, which are actually the ones requiring parallel processing, the overhead of inspector step is later amortized during the executor step. Our experiments show that for a graph of 16M vertices, the inspector-executor approach is 3.5 times faster than the original implementation, and thus, this optimization was used in the CPU Top-Down implementation. With respect to the CPU Bottom-Up implementation, a *parallel for*, i, traverses all the vertices, $v[i]$, but only when $dist[i] == \infty$ the corresponding neighbors, $e[v[i]]...e[v[i+1]-1]$, are checked. As soon as one of the neighbors is found in the old frontier, the vertex $v[i]$ is added to the new one by updating $bm_{new}[i] = true$.

The GPU and heterogeneous `Selective` and `Concurrent` implementations use OpenCL 1.2. In the case of the OpenCL Top-Down kernel the inspector-executor approach does not pay off due to three reasons:
i) the cost of the inspector step to generate the dense frontier; ii) consecutive vertices in the dense frontier processed by the same GPU warp contain non-coalesced memory accesses that are slower to process due to memory divergence; and iii) the amount of parallelism in a dense frontier is reduced with respect to processing the original frontier. We have experimentally validated that the best performance is achieved when the GPU process that original (sparse) frontier. Although with this approach some of the threads of the GPU warp end up doing nothing, this control divergence is not as expensive as the memory divergence of the inspector-executor approach. The Bottom-Up OpenCL kernel is essentially equivalent to the CPU Bottom-Up implementation.

Notice that OpenCL 1.2 version does not allow coherent read/write operations over the same memory locations from both GPU and CPU. However, in our `Concurrent` implementation, during the CPU-GPU Bottom-Up phase, the CPU and GPU write in different non-overlapping regions of the output vectors ($dist$ and bm_{new}) so this is not a problem.

V. Experimental Evaluation

A. Environmental setup and input graphs

The following platforms were used to run the experiments:

- ODROID-XU3: Embedded/mobile platform with a Samsung Exynos5422, 4 Cortex-A15 2Ghz and 4 Cortex-A7 CPU cores, a Mali-T628 MP6 GPU, and 2GB of LPDDR3 RAM. This is a development platform for mobile, as it has similar characteristics as most modern mobile phones. It runs Linux Ubuntu 14.04 and GCC 4.8.
- Intel® Core™ i7-4790: 4 cores at 3.60GHz, 16 GB 1666 MHz DDR3, and an integrated GPU Intel® HD Graphics 3000. It runs Linux CentOS 7.0 and GCC 4.8.
- Intel® Core™ i5-4250U: 2 cores at 1.3 GHz, 4 GB 1600 MHz DDR3, and an integrated GPU Intel® HD Graphics 5000. It runs MacOS X El Capitan and clang-omp 3.5.

In the following sections, we will refer to them as Odroid, Core i7, and Core i5, respectively. Table I shows the graphs used in our experiments. Well-known sources of graphs were used: SNAP collection [9], Graph500 graphs generators [10], and synthetic generated graphs from Rodinia Benchmark [11]. These sets of inputs provide a wide range graphs sizes and structures, including social network and roads graphs.

B. Performance and energy efficiency evaluation

An experimental evaluation was made over different platforms in order to understand the impact of the approaches on performance and energy efficiency. Figures 2 and 3 show the performance and energy evaluations. Performance results are shown using the metric *M*illions of Traversed Edges Per Second (MTEPS) [7]. This metric, as used in the literature, results from dividing the number of edges in a graph by the execution time, thus it provides a normalization of performance across graphs with different number of vertices and edges, and therefore, different running times. For energy efficiency, the metric used was performance per watt, or Millions of Traverse Edges per Joule (MTEPJ). Power is measured using libraries provided by the vendors (Intel® PCM and an in-house library that access power dissipation readouts in the Odroid). For collecting the results, the BFS algorithm was run 500 times from different source vertices, s_0, and the average was calculated.

In the figures, we show the evaluation when BFS is configured with the method Top-Down and launched on CPU (`CPU TD`) or on GPU (`GPU TD`), and when BFS is configured with the method Bottom-Up and launched on CPU (`CPU BU`). Results for the Bottom-Up method on GPU are not shown because the first iterations are too slow, so the executions take too long. Also, for the performance evaluation and to establish a baseline, we compare our approaches

Abbreviation	Description	# Vertices x10e6	# Edges x10e6	Source
amazon	Amazon product network	0.33	0.92	SNAP
youtube	Youtube online social network	1.13	2.98	SNAP
google	Web graph from Google	0.875	5.105	SNAP
liveJournal	LiveJournal online social network	3.99	34.68	SNAP
wiki-talk	Wikipedia talk (communication) network	2.39	5.02	SNAP
email	Email network from a EU research institution	0.26	0.42	SNAP
graph4M	Rodinia Generated	4.19	25.16	Rodinia
graph8M	Rodinia Generated	8.38	50.34	Rodinia
graph16M	Rodinia Generated	16.77	100.62	Rodinia
kron	Graph500 Kroneker	1.75	41.10	Graph500
rmat	Graph500 Rmat	2.13	81.16	Graph500
roadCA	Road network of California	1.96	2.76	SNAP
roadTX	Road network of Texas	1.37	1.91	SNAP
roadPA	Road network of Pennsylvania	1.08	1.54	SNAP

TABLE I
LIST OF GRAPHS

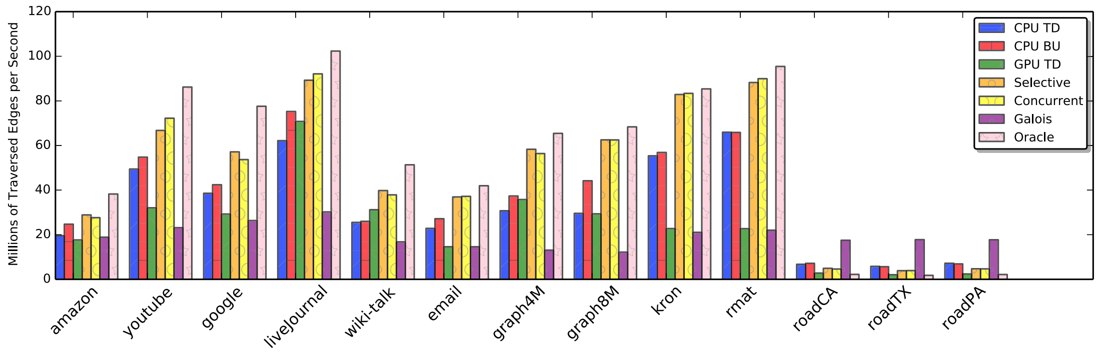
named `Selective` and `Concurrent`, against an `Oracle` execution which consider both methods (TD, BU) and devices (CPU, GPU). In particular, for each graph we collected the time that each method spent on each iteration (or frontier) when running on each device. Then, the oracle is calculated by taking, for each iteration, the minimum execution time for each method and device, and adding up those minimums for all the iterations. Thus, ideally the oracle represents the minimum execution time when the best method and device is chosen for each iteration, without incurring any overhead. Also, for the performance results another baseline is considered: `Best Galois`. For each graph, it represents the best result obtained among the different implementations that Galois offers for BFS [4]. Galois is a well-optimized and state-of-the-art suite of different algorithms for graph applications, which makes efficient use of all CPU cores present in the system, so these results represent a reference for the CPU executions.

Figure 2(a) shows the performance results obtained after running different graphs on the Odroid platform. As we see, the `Selective` and the `Concurrent` heterogeneous approaches work better than the implementations where only one device is used. An average speedup of 1.41x and 1.40x was obtained for `Selective` and `Concurrent` approaches respectively, with speedups of up to 1.52x and 1.56x for the Rodinia graphs. The speedups have been calculated against the best of the only one device executions (`CPU TD`, `CPU BU` or `GPU TD`) for each graph. As we see, although the `Concurrent` approach uses both devices simultaneously, it only does so in some of the iterations. Although these iterations typically represent around 80% of the execution time, the improvement with regards `Selective` is small on some of the graphs (the larger ones), but on average it does not outperform `Selective`. In section V-C we further explore this issue pointing out that the congestion on the memory bus, when both devices are working concurrently, increases the number of stall cycles seen by both devices. This, in turn, degrades performance. On the other hand, an average speedup of 1.57x was obtained by `Oracle` which indicates that the overhead of the `Selective` approach is below 10%.

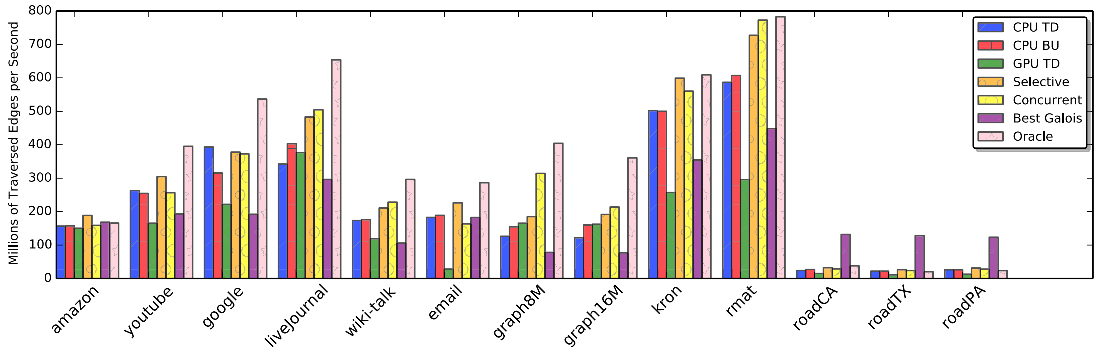
One interesting result is that Galois performed poorly when compared with the heterogenous approaches in all the social network graphs. However, Galois performed several times better in road networks. Let's remind that our approaches use CSR to represent the graphs, while Galois uses more general and complex data structures (sets and graphs) that take advantage of asynchronous graph traversals. The use of CSR provides the ability of exploring a frontier fast because of locality, but there is a need for synchronization after each iteration that Galois implementations may avoid. Because of this, graphs that are highly connected but need a small number of iterations like social network graphs, will benefit more from locality and will likely be less affected due to synchronization barriers when the data structure used is CSR. In the case of road networks with lowly connected graphs and a high number of iterations, the amount of work per iteration will be small and the synchronization overhead will be more significant in approaches based on CSR and thus these graphs will benefit from the asynchronous approaches implemented in Galois.

In the case of the Core i7 (Figure 2(b)), the results of the heterogeneous approaches are not as good as the ones obtained for Odroid, but this is an expected outcome as the GPU capabilities are not comparable to the computing power of the CPU cores. For this particular platform, an average speedup of 1.07x and 1.08x are achieved for the `Selective` and `Concurrent` approaches respectively, with the best results obtained by the Rodinia graphs with speedups of 1.14x and 1.46x. On this architecture, `Concurrent` performs better than `Selective` because the memory bus exhibits higher memory bandwith than on the Odroid, so it is not as congested when both devices work concurrently.

Another platform where we performed the same experiments is Core i5 (not shown due to space constraints). In this platform, both the CPU and GPU capabilities are bigger than in the Odroid. The GPU is also bigger in this case when compared with the Core i7 but the Core i5 has smaller CPU cores compared to the 4 high-end cores in the Core i7. Thus, it is a platform which potentially can benefit more from heterogeneous approaches. In fact, we find that improvements of the `Selective` and `Concurrent` approaches are 1.31x and 1.33x respectively, on average for all the graphs. Figure 2 summarizes the different performance improvements for all the platforms.



(a) Performance evaluation on Odroid



(b) Performance evaluation on Core i7

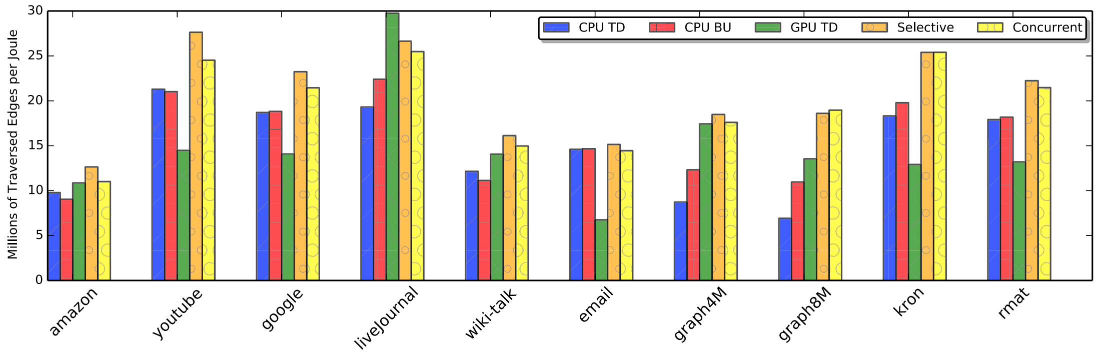
Fig. 2. Performance on Odroid and Core i7.

Another important factor to consider on all platforms is the energy efficiency obtained. Figure 3 shows how both the `Selective` and `Concurrent` heterogeneous approaches are on average 1.28x and 1.32x more energy efficient on the Odroid, and 1.23x and 1.17x more energy efficient on Core i7 when they are compared to any other approach that only uses one device. This energy efficiency can be attributed to two main factors. First, these approaches traverse the graph in less time than any other approach, thus draining less power for less time. Second, GPU at peak performance drains less power than the CPU cores under similar stress, and these approaches make more use of the GPU. However, we also notice that the `Concurrent` approach can be less energy efficient that `Selective` when the former can provide more performance (see for instance Youtube graph in Odroid or rmat graph in Core i7). This reinforce the fact that in heterogeneous computing not always minimum execution time results in minimum energy consumption. Figure 4(b) summarizes the energy efficiency improvements for all the platforms (including Core i5 where we get 1.21x and 1.23x of efficiency). Interestingly, on the Core i7 the improvement in energy efficiency is larger than in performance for the heterogeneous approaches. On the other hand, on the Odroid and Core i5, the energy efficiency of the `Concurrent` approach is slightly better than `Selective`. In any case, the differences between the heterogenous approaches are not big.

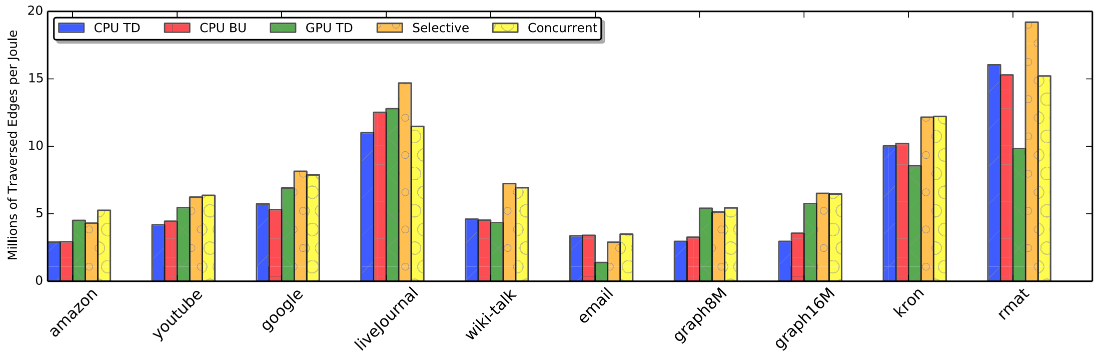
C. *Memory Bandwidth limitations*

Since heterogenous platforms provide a shared-memory environment, the data in memory can be used by both devices at the same time without requiring memory copies (`Concurrent` approach). This is why it is so appealing to explore algorithms in which both devices can traverse the graph in parallel. But after implementing the approach aimed to exploit this characteristic and finding no notorious improvement, a more comprehensive study was performed around this matter, in particular on the Odroid platform.

Figure 5(a) shows how the `Concurrent` approach behaves when both devices are working at the same time (`CPU+GPU`). This experiment consisted on running the iteration with the heaviest workload on both devices and measuring times when a different percentage of the workload is offloaded to the GPU while the CPU is working on the remaining load. The x-axis represents the different partitions considered. We also measured the time that each device would take to finish its part of the workload when working alone in the system (`CPU Only` and `GPU Only` results). The tests were performed using as input the Rodinia 16M graph. In the ideal case, the last bar is expected to be as big as the max of the first two. In other words, when both the CPU and the GPU are working, it is expected that all the work will be finished after the slowest device is done. But this is not the case, and when both devices work at the same time, the memory system bus cannot provide

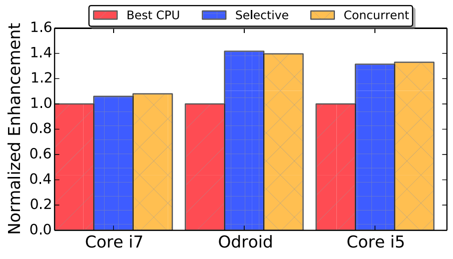


(a) Energy efficiency evaluation on Odroid

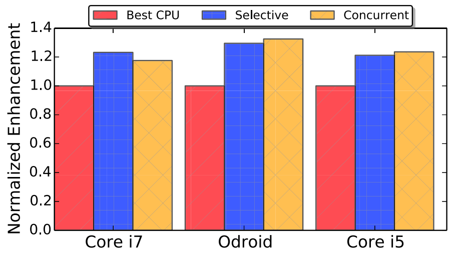


(b) Energy efficiency evaluation on Core i7

Fig. 3. Energy efficiency evaluation on Odroid and Core i7.



(a) Performance comparison



(b) Energy efficiency comparison

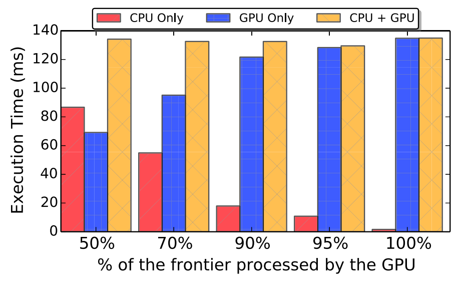
Fig. 4. Performance and Energy efficiency normalized for comparison between platforms.

the devices with the necessary memory bandwidth. When the number of memory requests per time increases because both devices are working at the same time, the bus gets congested, producing an important number of stall cycles on both devices and preventing the devices to work at their peak performance. We measured more than 40% of performance degradation in this case. This behavior is due to the nature of the BFS processing, which is heavily memory bound. To compare this benchmark with a similar but compute bound one, we added dummy operations on the BFS kernels (several square root calculations). Figure 5(b) shows how in the case of this new compute bound application, a `Concurrent` approach would have the potential to achieve close to ideal performance on the Odroid platform. Similar results were observed on the Core i7 and Core i5 platforms.

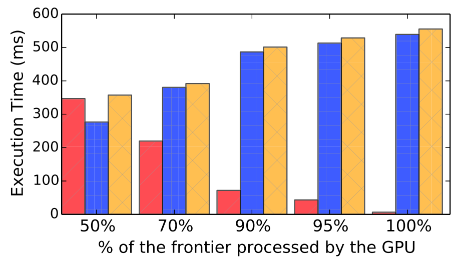
VI. Previous Work

Some research has been done on the BFS algorithm, but just few of them were aimed to run on a heterogeneous platform. There are algorithms specifically implemented for GPU, such as the Rodinia [11] and the Parboil [12] benchmarks, but, to the best of our knowledge, there are only a couple of implementations using both CPU and GPU.

One of them is a Task-Based approach [6], whose best performing implementation only uses one of the devices at a time, depending on the number of vertices to be processed.



(a) Memory Bound



(b) Compute Bound

Fig. 5. Comparison between memory bound (regular BFS) and compute bound (artificial BFS) in Odroid.

Hong et al. [13] is the other attempt to integrate CPUs and GPUs in a hybrid environment by choosing dynamically between three different implementations for each level of the BFS algorithm, but again, sending work to CPU or GPU and not using both at the same time. In both cases, the evaluated platform consist of a high-end GPU connected to the multicore via a PCIe bus for which the communication cost is too high to make it worth it. On the contrary, the aim of this work is to come up with algorithms that are more suitable for shared memory platforms where this communication cost are less harmful to the overall execution.

Beamer at al. [5] introduce a novel approach for performing BFS which introduces speedups compared to regular implementations, but again, it is aimed to multi-core environments. More related to our work is the approach proposed by Data et. al [7] which is similar to our `Selective` approach. However, in our work by taking advantage of the characteristics of the social network graphs, we simplify the heuristic to switch from the Top-Down CPU to the Bottom-Up GPU and reduce the overheads of GPU kernel launching by enqueuing three iterations (frontiers) after the switch, finding that the overhead of our approach compared to an ideal oracle execution is less than 10% . Additionally, we study a novel `Concurrent` approach in which both the GPU and the devices can be put to work simultaneously over the same frontier and evaluate its limitations.

VII. Conclusion and Future Work

This study focused on the implementation and evaluation of heterogenous approaches on the BFS algorithm for graph processing, achieving average speedups of 1.41x, 1.08x and 1.33x for the Odroid, Core i7 and Core i5 platforms and improvements in energy efficiency of 1.32x, 1.23x and 1.24x, respectively. Our studies led to the conclusion that the best performance and energy improvement can be obtained by having CPU cores and GPU with similar computing capabilities, which is an environment mostly present in hand-held devices and mobile platforms (like the Odroid).

Our approaches explored characteristics of social network graphs and attempted to transform these characteristics into opportunities for speedups in graph traversing, exploiting the shared-memory resources that heterogeneous platforms with integrated accelerators expose. However, given the memory bound nature of this application, the limiting factor in performance is the memory bus bandwith, which in case of congestion when both the CPU and GPU devices work concurrently, can severely degrade the peak performance of each device. The exploration of novel approaches to mitigate the congestion of the memory bus, as well as the study of other irregular algorithms for graphs is left as future work,

Acknowledgments

This work is supported in part by NSF under grant CNS-1319657 and by Spanish projects TIN2013-42253-P and P11-TIC-08144.

References

[1] T. Hiragushi and D. Takahashi, “Efficient hybrid breadth-first search on GPUs,” in *Intl. Conf. on Algorithms and Architectures for Parallel Processing*, 2013, pp. 40–50.
[2] D. G. Merrill and A. S. Grimshaw, “Revisiting sorting for GPGPU stream architectures,” in *PACT*, 2010, pp. 545–546.
[3] A. Munshi., “The OpenCL specification,” 2012. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
[4] G. I. T. U. of Texas at Austin, “http://iss.ices.utexas.edu/.”
[5] S. Beamer, K. Asanović, and D. Patterson, “Direction-optimizing breadth-first search,” in *Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 12:1–12:10.
[6] L. Munguia, D. Bader, and E. Ayguade, “Task-based parallel breadth-first search in heterogeneous environments,” in *Intl. Conf. on High Performance Computing (HiPC)*, Dec 2012, pp. 1–10.
[7] M. Daga, M. Nutter, and M. Meswani, “Efficient breadth-first search on a heterogeneous processor,” in *Conf. on Big Data*, Oct 2014.
[8] A. Bulu, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, “Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks,” in *IN SPAA*, 2009, pp. 233–244.
[9] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” http://snap.stanford.edu/data, Jun. 2014.
[10] Graph500, “http://www.graph500.org/.”
[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Symp. Workload Characterization, IISWC*, Oct 2009, pp. 44–54.
[12] L. Luo, M. Wong, and W.-m. Hwu, “An effective GPU implementation of breadth-first search,” in *DAC*, 2010, pp. 52–55.
[13] S. Hong, T. Oguntebi, and K. Olukotun, “Efficient parallel graph exploration on multi-core CPU and GPU,” in *PACT*, 2011, pp. 78–88.