On Graphs, GPUs, and Blind Dating
A Workload to Processor Matchmaking Quest

Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, Matei Ripeanu
Department of Electrical and Computer Engineering, The University of British Columbia
{abdullah, lauroc, elizeus, matei}@ece.ubc.ca

Abstract— **Graph processing has gained renewed attention. The increasing large scale and wealth of connected data, such as those accrued by social network applications, demand the design of new techniques and platforms to efficiently derive actionable information from large scale graphs.**

Hybrid systems that host processing units optimized for both fast sequential processing and bulk processing (e.g., GPU-accelerated systems) have the potential to cope with the heterogeneous structure of real graphs and enable high performance graph processing.

Reaching this point, however, poses multiple challenges. The heterogeneity of the processing elements (e.g., GPUs implement a different parallel processing model than CPUs and have much less memory) and the inherent irregularity of graph workloads require careful graph partitioning and load assignment. In particular, the workload generated by a partitioning scheme should match the strength of the processing element the partition is allocated to. This work explores the feasibility and quantifies the performance gains of such low-cost partitioning schemes.

We propose to partition the workload between the two types of processing elements based on vertex connectivity. We show that such partitioning schemes offer a simple, yet efficient way to boost the overall performance of the hybrid system. Our evaluation illustrates that processing a 4-billion edges graph on a system with one CPU socket and one GPU, while offloading as little as 25% of the edges to the GPU, achieves 2x performance improvement over state-of–the-art implementations running on a dual-socket symmetric system. Moreover, for the same graph, a hybrid system with dual-socket and dual-GPU is capable of 1.13 Billion breadth-first search traversed edge per second, a performance rate that is competitive with the latest entries in the Graph500 list, yet at a much lower price point.

I. INTRODUCTION

Two reasons make us believe that commodity hybrid systems (i.e., GPU-accelerated nodes) are an appealing platform for high-performance, low-cost graph processing. First, Graphical Processing Units (GPUs) bring massive hardware multithreading able to mask memory access latency – the major barrier to performance for this class of problems. Second, a hybrid system that hosts both processing units optimized for fast sequential processing and units optimized for bulk processing matches well the heterogeneous structure of the graphs that need to be processed in practice, as these graphs have variable amounts of parallelism – whether across the execution phases of an algorithm or across the workload itself (e.g., scale-free graphs).

The challenge to materialize these opportunities, however, is that the graph must be partitioned and the resulting partitions must be assigned to processing elements.

Matching graph workload partitions to processors while disregarding their intrinsic characteristics can be as good as blind dating. One can have surprisingly good outcomes, yet little assurance in terms of the quality of the match. This work sheds light on the intrinsic characteristics of both graph workloads resulting from partitioning and processing engines in a hybrid platform (i.e., CPUs and discrete GPUs) that enable principled workload-to-processor matchmaking.

In other words, this paper makes progress on answering the following high-level questions:

- Is it possible to partition the graph and the processing tasks to efficiently use both the traditional CPU cores and GPU(s)? More specifically, are there low-complexity partitioning scheme algorithms that generate partitions that match well the individual strengths of CPU and GPU processing elements?
- For a given class of graph problems and a fixed-power budget or cost, what is the optimal balance between traditional and massively-parallel processing elements (e.g., should one assemble a machine with four CPUs or the same performance can be obtained with one CPU and one GPU)?

Making progress on answering these questions is both difficult and important. On the one side, making progress on answering these questions is *important* not only because progress may unlock the knowledge gained through the ability to process large graphs faster and at lower cost. More importantly, this problem is an instance of a general problem that is highlighted by current hardware trends: As the relative cost of energy continues to increase relative to the cost of silicon, future machines/processors will host a wealth of different computational units and the key issue will become decomposing the workload such that various workload partitions are assigned to the processing unit where they can be executed most efficiently (e.g., for power or time) [9].

On the other side, multiple factors make answering these questions *harder* than in the context of homogeneous platforms and more regular workloads. Among these we list the heterogeneity of computational models of the processing elements that compose hybrid systems, and the fact that the existing parallelism is data dependent.

The concrete contribution of this work is fourfold:

- We propose a set of low-cost graph partitioning and workload assignment strategies (Section IV) that are tailored for parallel graph processing on hybrid systems. Our strategies are informed by vertex-connectivity and successfully partition the graph such that the workload assigned to the bottleneck processing element exploits well its strengths leading to performance gains that scale supra-linearly with the share of the workload assigned to the bottleneck engine.
- We put forward a set of *practical guidelines* for choosing the appropriate partitioning and workload assignment strategy. The choice depends on the particular deployment (memory size of processing elements) and problem characteristics (graph size, computational intensity, cache friendliness). (Section VII)
- We present a detailed evaluation of the performance impact of the graph partitioning and workload assignment strategies we propose. We scan over one order of magnitude of graph sizes (from 0.5 to 8 billion edges) and five hardware configurations, explain the reasons for the observed performance in detail (e.g., using hardware counters), and we use two graph algorithms Breadth-First Search and PageRank that stress the computational platform in different ways. (Section V)
- Finally, we demonstrate good raw-performance numbers using graph algorithm implementations on top of an algorithm agnostic graph-processing platform (TOTEM - which, although not the focus of this paper, has benefited from renewed attention as a result of our experiments). Not only these performance numbers match best published results on similar hardware, but they are competitive with (though at the bottom of) recent entries in the Graph500 challenge. For example, on a hybrid system with dual-socket and dual-GPU our implementation delivers 1.13 Billion breadth-first search traversed edges per second for a scale-28 RMAT graph. (Section V.E).

II. BACKGROUND

This section discusses our own prior work [13], which this study builds on, presents the graph computational model we assume (Bulk Synchronous Parallel), and introduces TOTEM, the graph-processing engine used in the previous and current studies.

A. Relationship With Our Prior Work

We have previously demonstrated the feasibility of graph processing on hybrid systems [13]. In particular, we focused on hybrid systems that use discrete GPUs (i.e., systems in which the CPU and the GPU memory are connected via a communication bus rather than sharing the same memory system).

First, our performance modeling shows that keeping communication overhead low is crucial to process graphs on hybrid systems. Second, informed by the performance model, we designed and implemented TOTEM, a generic graph-processing engine for single-node hybrid systems.

TOTEM uses a number of algorithm-agnostic optimizations. Out of these, the most important is reducing the communication overhead by over an order of magnitude by aggregating messages sent from one processing element to another (e.g., GPU to CPU). As we discuss later, reducing the communication overhead has a practical and important impact on the design of graph workload partitioning strategies. In summary, this optimization eliminates the need to focus on minimum cuts when partitioning the graph (a pre-processing step that is prohibitively expensive).

We validated the performance model and, more importantly, we showed that, compared to processing only on the CPU, using a GPU is feasible to obtain near linear speedups with respect to the proportion of the graph that is offloaded to the GPU [13].

Our previous work, however, was limited to random partitioning and small-scale graphs. This work explores the effectiveness of various graph partitioning strategies and workload allocation schemes on the performance of graph algorithms on a hybrid system. We focus on investigating low-cost partitioning techniques and to generated workloads that match well the strength of the processing element they are allocated to. *Additionally, we evaluate our techniques on new state-of-the-art hardware and use significantly larger graphs (at multi-billion edges scale).*

We implemented our partitioning strategies in TOTEM; hence, the next subsection presents its computation model.

B. TOTEM Computation Model

TOTEM adopts a Bulk Synchronous Parallel (BSP) [27] computation model. In BSP, processing is divided into rounds (*supersteps* in BSP terminology), each consisting of three phases executed in order: computation, communication and synchronization. In the computation phase, each processing element (the CPU and the GPU(s)) executes computations asynchronously on its local memory. In the communication phase, the processors exchange messages that are necessary to update their status before the next computation phase starts. The synchronization phase guarantees the delivery of the messages.

Figure 1 shows how BSP is used in TOTEM. The developer specifies a set of callback functions. TOTEM partitions the graph, transfers the partition to the GPU, and launches the computation according to the callbacks provided by the developer. Both processors, CPU and GPU, execute the user defined `algo_compute_func` on their own

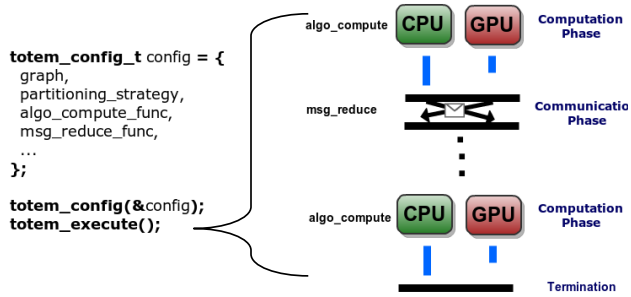

Figure 1: A simplified TOTEM configuration and how an algorithm callbacks map to the BSP execution model.

partitions concurrently (computation phase). Then, TOTEM uses `msg_reduce_func` to aggregate the messages sent to the same remote vertex, and bulk transfer messages to the destination processor (finishing a round/ BSP *superstep)*. Invocations of `algo_compute_func` and `msg_reduce_func` continue for each round until the algorithm converges to termination.

Using the BSP model offers two major advantages. First, it provides a simple framework to implement graph algorithms on distributed memory systems. Second, and more importantly, it allows circumventing the high-latency of the PCI-Express bus by batching data transfers in the communication phase.

III. WORKLOAD AND PROCESSORS CHARACTERISTICS

This section discusses the characteristics of both graph workloads and hybrid systems.

A. Characteristics of Graph Workloads

Graph workloads and algorithms exhibit the following characteristics:

(i) Modest processing requirements per vertex in each round. A typical algorithm processes a graph in rounds and, in each round, only a subset of vertices may be active and may be processed in parallel. For example, in Breadth-first Search (BFS), each vertex iterates over its neighbors attempting to set their depth. Similarly in PageRank, each vertex computes a new rank by accumulating the ranks of its neighbors.

(ii) Poor locality. The neighbors of a vertex are scattered in memory, therefore iterating over them results in accesses that are not spatially coherent and renders graph workloads memory bound.

(iii) Imbalanced workload distribution. Many relevant real graphs have power-law degree distribution: a few vertices will have many edges and most vertices only one or a few edges. Examples of such graphs include social networks [2], the Internet [10], the World Wide Web [4], financial networks [17], protein-protein interaction networks [18], semantic networks [26] and airline networks [28] to mention few. The skewed distribution of edges in real-world graphs leads to imbalanced workload distribution across vertices, where the high-degree vertices imply larger processing tasks.

B. Characteristics of Hybrid Systems

Table 1 lists the specifications of state-of-the-art CPU and GPU models that we use in our experiments. The two processing elements are characteristic for their categories and support different performance attributes. On the one hand, GPUs have significantly larger number of hardware threads, higher memory access bandwidth, and support a larger number of in-flight memory requests. On the other hand, the CPU cores are clocked at over double the frequency, and have access to roughly one order of magnitude larger memory and cache.

To cope with the graph's memory-bound workload, the GPU relies on its ability to launch hundreds of threads and in-flight memory requests to hide memory access latency. However, the GPU's limited memory space means it can process only relatively small graphs (or graph partitions).

Table 1: Specifications of state-of-the-art CPU and GPU models used in our experiments.

Characteristic	Intel Nehalem Xeon X5650	Fermi GPU Tesla C2075
Core Frequency	2.67GHz	1.15GHz
Num Cores	6	14
HW-thread/Core	2	32
Last Level Cache	12MB	2MB
Main Memory	148GB	6GB
Memory Bandwidth	32GB/sec	144GB/sec
Total Power (TDP)	95W	225W

The CPU, however, can process an order of magnitude larger graphs as it has access to significantly larger memory space. To hide memory access latency, the CPU employs a larger cache. Some algorithms attempt to achieve high cache hit rates by leveraging summary data structures. BFS can, for example, keep track of which vertices have been visited in a bit-vector (more on this in Section IV.C).

IV. GRAPH PARTITIONING FOR HYBRID SYSTEMS

This section presents the set of features required for effective partitioning strategies for hybrid systems, details our proposed degree-based partitioning strategy, and discusses how this strategy aligns with the required features.

A. Required Features for Graph Partitioning Strategies

We define the following general characteristics for an effective partitioning strategy:

- *Has a low space and time complexity*. Processing large-scale graphs is expensive in terms of both space and time; hence we believe that partitioning algorithms that have time complexity higher than linear or quasilinear are impractical.
- *Handles scale-free graphs.* Many important networks in different domains present skewed vertex degree distributions. Therefore, the partitioning strategy must be able to handle the severe workload imbalance that is associated with such networks.
- *Handles large (billion-edge scale) graphs.* The amount of memory offered by single-node systems is considerably large; for instance, 144GB on our evaluation machine is enough to fit a graph with half billion vertices and eight billion edges (i.e., a scale-29 graph in Grap500 terminology).
- *Minimizes algorithm's execution time by reducing computation* (rather than communication). The BSP model divides processing into computation and communication phases. We focus on partitioning strategies that lead to lower computation time. We note that our approach is in sharp contrast to previous work on graph partitioning for distributed graph processing, as they focus on minimizing the time spent on

```
BFS_kernel(Partition partition, int level) {
  bool finished = false;
  parallel for v in Partition.vertices {
    if (v.level != level) continue;
    for (n in v.neighbors) {
      if (!Partition.visited.isSet(n)) {
        if (Partition.visited.atomicSet(n)) {
          n.level = level + 1;
          finished = false;
  }}}}
  return finished;
}
```

Figure 2: Pseudocode for level-synchronous Breadth-First Search (BFS) compute kernel [16]**. The kernel is invoked in each round for each partition. The algorithm terminates when all partitions in the same round return true. Note that if the graph is processed by one processor, the whole graph is one partition.**

communication (e.g., by minimizing the edge-cut [1] between partitions) [6]. Our past experience [13] provides the intuition that supports this choice: the communication overhead for concurrent graph processing (or similar applications, as the optimization is application agnostic) on hybrid systems can significantly be reduced via message aggregation and batch communication (assisted by the high bandwidth of the PCI-E bus that typically connects discrete GPUs), and hence computation rather than communication is the bottleneck. We confirm this intuition with our evaluation in Section V.

B. Partitioning Strategy: Partition by Vertex Degree

We propose to partition the graph by placing the high-degree vertices in one type of processor and the low-degree ones in the other type. Our hypothesis is that this simple and low-cost partitioning strategy brings tangible performance benefits.

The motivation behind this idea is twofold. First, dividing a real-world graph by using the vertex degree as the partition criterion produces partitions with significantly different degrees of parallelism that match those of the different processing elements of the hybrid system. Second, such a partitioning strategy produces partitions that are more homogenous in terms of vertex connectivity compared to the original graph, resulting in a more balanced workload across the vertices of a partition. This is important to maximize the utilization of a processor's cores, especially for the GPU, which has strict parallel computing model.

Partitioning the graph based on vertex degree is low cost in terms of computational and space complexity. One way to classify the low and high degree vertices is by sorting, which has quasilinear time complexity of O(|*V*|**log**|V|). However, we do not even need to completely sort the vertices, as we can do well enough with partial sorting (i.e., finding the degree values that divide the graph into the desired partitions), which takes linear O(|*V*|) time complexity [7]. Additionally these manipulations can be done in place. This is important when handing large-scale graphs.

[1] The *edge-cut* is the number of edges cut as a result of splitting the graph into partitions.

```
PageRank_kernel(Partition partition) {
  parallel for v in Partition.vertices {
    double sum = 0;
    for (n in v.neighbors) {
      sum += n.rank;
   }
   v.rank = 0.15/gVertex_count + 0.85*sum;
  }
}
```

Figure 3: A simplified PageRank compute kernel. The kernel is invoked in each round for each partition. The algorithm terminates after executing the kernel a predefined number of times [22]**.**

C. Two Case Studies: BFS and PageRank

To shed more light on the effect of such partitioning strategy, we consider two example algorithms: Breadth-First Search (BFS) and PageRank.

Figure 2 presents a BFS kernel that is based on the level-synchronous algorithm. Hong et al. [16] showed that for low-diameter graphs, this implementation has a superior performance compared to the typical queue-based approaches. Queue-based approaches strive to be work efficient by maintaining the frontier vertices in a queue to avoid iterating over all the vertices in each round. However, for low-diameter graphs, the approach proposed here works better, as it manages to utilize memory bandwidth more efficiently for both, CPU and GPU implementations [16].

In order to reduce main memory traffic, the algorithm uses a bit-vector (named "visited" in Figure 2) to mark the vertices that have already been assigned a level, avoiding fetching their state from main memory.

Chhugani et al. [8] showed that a cache-resident "visited" bit-vector is critical for BFS performance on the CPU, and that the performance significantly drops for large graphs. This is because the bit-vector becomes larger than the LLC, hence the cache hit rate is significantly reduced, and as a result the overall performance.

Consider for example a scale-free graph with 256M vertices and 4B edges. If each vertex is represented by one bit, the bit-vector length is 256M/8=32MB, which does not fit in the 12MB of the LLC of our CPU model (see Table 1), or even a dual-socket configuration (which would have 24MB of LLC).

For this workload, offloading the many low-degree vertices to be processed concurrently on the GPU shrinks the bit-vector size of the CPU partition considerably. As we show in the evaluation (Section V.B), offloading the low-degree vertices to the GPU enables a significantly more efficient use of the CPU resources, and consequently enables higher performance. Note that many low-degree vertices can be offloaded to the memory-limited GPU because they are connected to a small number of edges, and hence their associated memory footprint is low.

However, not all algorithms are able to use summary data structures the same way as BFS does. In PageRank, for example, where every vertex must access the full state of all its neighbors in order to calculate a new rank, a summary data structure is not sufficient. Figure 3 shows the compute kernel of the PageRank algorithm.

Nonetheless, vertex connectivity-based partitioning is still beneficial for algorithms like PageRank. Using the same strategy as above, placing the many low degree vertices on the GPU and keeping the fewer high degree ones on the CPU matches the level of parallelism offered by each processing element: the highly parallel workload is handled by the GPU and the part that has less parallelism is handled by the few, but powerful CPU cores. Moreover, as mentioned before, there is an additional benefit: the workload assigned to each processing element is more balanced across its cores.

V. EVALUATION

Goals. The main goal of our evaluation is to test the central hypothesis of this paper that *partitioning graph workloads while taking into account the intrinsic characteristic of both the workload and the processors lead to better performance*. To this end, we conduct a detailed set of experiments to quantify and analyze the impact of degree-based partitioning strategies on the performance of the hybrid system. More specifically, this section aims at addressing the following questions:

- *How do different partitioning and placement strategies influence the performance of graph processing on a hybrid system?* CPUs and GPUs have different performance characteristics; hence graph partitioning and data placement strategies play a critical role in the performance of the hybrid system.
- *What is the effect of the algorithmic cache friendliness and the compute-to-memory ratio on the performance of the hybrid system?* The compute-to-memory access ratio and caching friendliness vary widely among graph algorithms. We examine how these factors affect the performance of a hybrid system and the way the graph should be partitioned to achieve best performance.
- *What is the effect of graph's vertex degree distribution on the performance of graph algorithms on a hybrid system?* In particular, we compare typical real-world graphs, which exhibit power-law distribution, and graphs with uniform vertex degree distribution.
- *How is the hybrid system performance compared to a symmetric one? Also, how does the system scale when increasing number of GPUs and/or CPU cores?*

Workloads. Unless otherwise noted, we use graphs with skewed degree distributions similar to those generated for the popular Graph500 benchmark (www.graph500.org). As for the benchmark, these graphs are generated using the Recursive MATrix (RMAT) process [5] with the following parameters: (A,B,C) = (0.57, 0.19, 0.19) and an average vertex degree of 16. We describe the graphs by the logarithm base two of their number of vertices, a notation used by the benchmark as well (e.g., a graph with $|V| = 2^{25}$ vertices is referred to as a "scale-25" graph and has $|E| = 2^{29}$ edges).

We believe that using RMAT as the workload generator is sound for two reasons. First, it is adopted by today's widely accepted Graph500 benchmark [14]. Second, RMAT graphs have similar characteristics to real-world graphs: they have a low-diameter and a highly heterogeneous vertex degree distribution (i.e., their vertex degree distribution is 'power-law') [3][2].

We evaluate the performance of BFS and PageRank as described in Section IV. For each evaluation point, we measure only the algorithm's execution (i.e., we do not include TOTEM's initialization overhead, which mainly consists of reading the graph from disk and building the graph data structure in memory). We present the average and 95% confidence interval over 64 runs (with randomly chosen seeds in the case of BFS).

Note that our goal is to draw conclusions for a wider set of algorithms, and not only for BFS and PageRank. However, we start with two hard ones that exhibit different characteristics at the extremes of the computational intensity and cache friendliness: BFS is a lightweight graph algorithm that is sensitive to cache utilization and communication overhead, while PageRank is more compute intensive and less sensitive to cache size and communication overheads.

Testbed. The experiments use a machine with dual-socket Intel Nehalem (X5650) and two Tesla C2075 NVIDIA GPUs (Table 1 details the specifications of the processing elements). The machine runs Fedora14, CUDA release 4.1 and driver version 64-285.05.33. The code is compiled using g++ 4.5.1 with "-O3" option. Finally, OpenMP was used to parallelize the CPU code.

A. Highlighting the Effect of the Partitioning Strategies

We evaluate three partitioning strategies: RAND, HIGH, and LOW. The first, RAND, divides the graph randomly. The other two strategies are based on degree centrality: HIGH-degree divides the graph such that the highest degree vertices are assigned to the CPU; and LOW-degree divides the graph such that the lowest degree vertices are assigned to the CPU.

Figure 4 shows BFS traversal rate in billion of traversed edges per second (TEPS) for a scale-28 RMAT graph ((|V|, |E|) = (256M, 4B)). Note that the graph is too large to fit entirely on the GPU and the host must keep at least 70% of the graph's edges.

In this figure, the *x*-axis represents the various edge percentages assigned to the CPU partition. For example, consider the 70% data point and HIGH partitioning. The vertices are sorted in decreasing order of their degree and assigned to the host in this order until 70% of the edges of the graph and their corresponding vertices are placed on the host. The remaining vertices and their edges are placed on the GPU. Similarly, in the case of LOW partitioning, the vertices are sorted in increasing order according to their degree and assigned to the host until it holds 70% of the edges.

[2] Although R-MAT has the properties mentioned here its widespread adoption brought intense scrutiny. Seshadri *et al.* [25] note that it can only generate lognormal tails and does not produce high clustering coefficients, in addition to a high number of unconnected vertices that needed to be filtered out. As part of a deeper future investigation with other workloads, we plan to use graphs generated by the model proposed by Seshadri *et al.* [25], which should address R-MAT shortcomings, as well.

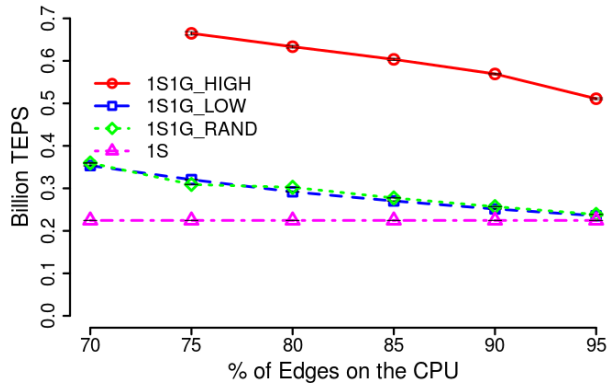

Figure 4: BFS traversal rate (measured in billion of edges traversed per second - TEPS) for a scale-28 R-MAT graph on a hybrid, one CPU socket and one GPU (1S1G), system for different partitioning algorithms while varying the percentage of edges placed on the host. For comparison, the performance of a single CPU socket (1S) processing the whole graph is also shown as a straight line.

Figure 4 demonstrates two points: First, compared to processing the graph on the host only, offloading part of the graph to be processed on a GPU is beneficial. This corroborates our previous results [13]. Second, and more surprisingly, the figure reveals a significant performance difference (higher than 2x for some cases) generated by the various partitioning schemes. The next subsection explores the causes for this performance difference.

B. Exploring the Causes of the Performance Difference

Figure 5 presents the breakdown of execution time for the same workload presented in Figure 4. The breakdown shows that the hybrid system's performance is bottlenecked by the CPU regardless of the partitioning scheme. This happens because of two reasons: (i) GPU is faster, since it has a higher processing rate and it can hold only a smaller partition

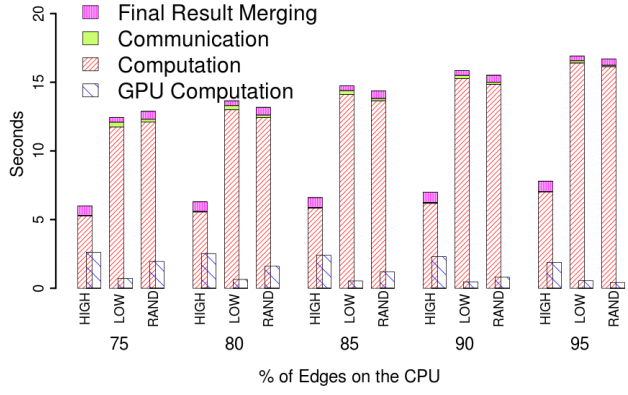

Figure 5: Breakdown of execution time for a scale-28 R-MAT graph. Note that the "Computation" bar refers to the computation overhead of the bottleneck processor (the CPU in this case as the computation phase is dominated by the processing of the CPU partition). The time to process the GPU partition, processed concurrently, is shown for comparison.

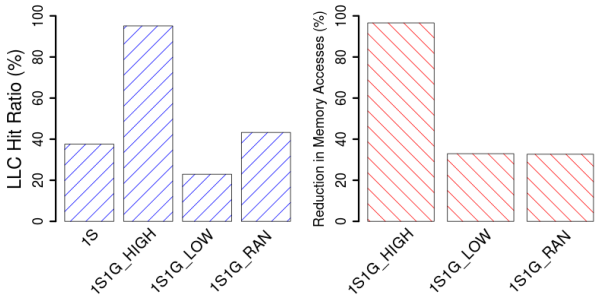

Figure 6: Exploration of the cache-hit rate. Performance counter statistics gathered for BFS for a scale-28 RMAT graph, when 75% of the edges are assigned to the CPU. *Left:* LLC hit ratio, computed as *100*LLC_HIT/(LLC_HIT+LLC_MISS)*. *Right:* the reduction in the number of main memory accesses on the host when compared to processing the whole graph on the host (using only one socket) computed as *100* ((LLC_MISS$_{1S}$ - LLC_MISS$_{1S1G}$)/ LLC_MISS$_{1S}$)*.

than the CPU due to memory constraints; and (ii) the other overheads (communication and final result aggregation) are negligible compared to the computation phase. Based on these two observations, the rest of this section focuses on the effect of graph partitioning strategies on CPU performance.

As discussed previously [8], one of the most important performance factors for BFS is the cache hit rate for the O(|*V*|) "visited" bit-vector. For this workload (scale-28 R-MAT graph), the size of the "visited" bit-vector is 32MB (i.e., a bit array that represents the 256M vertices) and it is over 2.5 times larger than the last level cache (LLC), which is 12MB.

Figure 6 shows the LLC cache hit rate for the different partitioning schemes when 75% of the edges are assigned to the CPU partition (left plot); and, for the same data point, the reduction in LLC misses in the hybrid system compared to the LLC misses when processing the whole graph only on a single CPU socket.

Depending on the partitioning strategy the "visited" vector is distributed differently between the host and the accelerator. Thus, to better understand the profiling data in Figure 6, Figure 7 shows the percentage of vertices assigned to the CPU due to graph partitioning. The two figures highlight the strong correlation between $|V_{cpu}|$ and the cache hit rate.

On the one hand, the RAND and LOW partitioning strategies produce a CPU partition with a large number of vertices leading to a "visited" vector comparable in size to that of the original graph. Therefore the LLC hit rate changes only slightly when compared to a single CPU socket: improved for RAND due to lower $|V_{cpu}|$, and worsened for LOW due to the added overhead of handling boundary edges (i.e., edges with source and destination vertices reside on partitions that are assigned to different processors). However, Figure 6 (right) shows that these strategies still reduce the number of main memory accesses by almost 30% – as a consequence of offloading part of the graph to the GPU –, resulting in the overall performance improvement brought by the hybrid system.

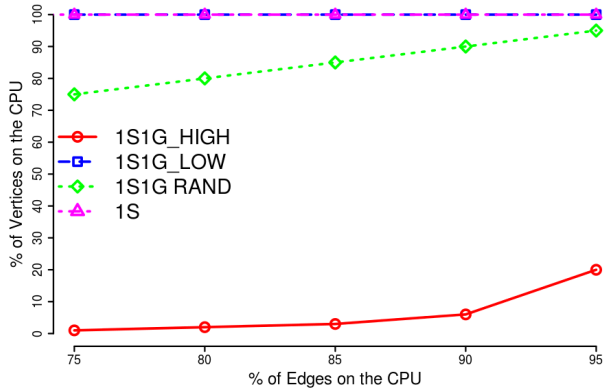

Figure 7: The percentage of vertices placed on the CPU for a scale-28 RMAT graph for various partitioning strategies. Note that the 1S and 1S1G_LOW lines overlap.

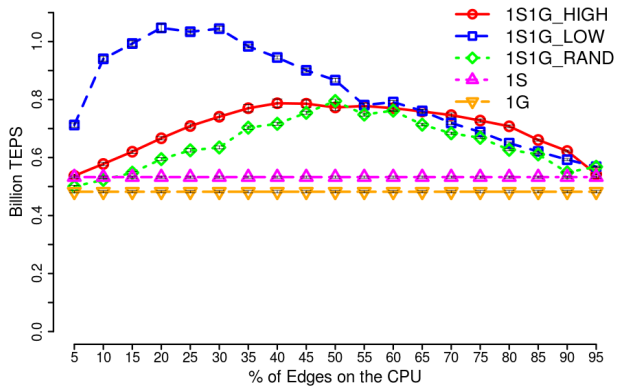

Figure 8: BFS traversal rate for a scale 25 R-MAT graph.

On the other hand, due to the power-law degree distribution of the graph, the CPU partition produced by the HIGH strategy has two orders of magnitude fewer vertices for the same number of edges, resulting in a much more cache friendly workload. This leads to a significant improvement in the CPU processing rate; as a result, the performance of the hybrid system is 2x faster than the other two partitioning strategies.

With the HIGH partitioning strategy, offloading as little as 5% of the edges to the GPU offers 2x speedup compared to processing the graph on the CPU only, and up to 3x speedup when offloading 25% of the edges. This demonstrates that although the GPU has limited memory, it can significantly improve the performance. *This is because the GPU is able to efficiently handle the sparser part of the graph as it relies on massive multi-threading rather than caches to hide memory access latency*. Indeed, Figure 5 shows that the computation phase for HIGH strategy takes longer on the GPU than for the other two strategies, although still not completely balanced between the GPU and the CPU.

Same exploration for a smaller graph. Figure 8 shows BFS traversal rate for a smaller, scale-25, R-MAT graph ((|*V*|,|*E*|) = (32M, 512M)). A smaller graph creates two different implications: it enables offloading a larger partition to the GPU (in this case, the graph can fit entirely in the GPU memory); and, for BFS, a graph with a small number of vertices improves cache hit rate of the algorithm (in this case, the bit-vector size is 4MB, and fits entirely in the 12MB of the LLC).

In the right end of the figure, where the CPU partition is larger than the GPU partition, CPU processing is the bottleneck and the performance of the three strategies exhibits the same behavior. Thus, the performance of the hybrid system is proportional to the share of the graph offloaded for processing on the GPU. Note that, because the graph is small and already fits the cache, the beneficial effect of the HIGH partitioning strategy is marginal compared to its effect on the larger scale-28 graph.

To understand the behavior of the hybrid system when most of the graph is processed on the GPU (the left end of Figure 8), Figure 9 shows the breakdown of execution time for the first data points. When partitioning the graph using RAND and HIGH strategies while keeping only a small percentage of the edges on the CPU, the result is a larger GPU partition with similar characteristics to the original graph. As Figure 9 shows, GPU processing is the bottleneck for both of these strategies and the gain brought by the hybrid system is proportional to the part of the graph processed concurrently on the CPU. Note that, for both strategies, the performance improves up to a point where the load is completely balanced between the two processing units. After that, it drops as the CPU partition becomes the bottleneck as discussed previously. Because in this workload GPU and CPU process the graph at almost the same rate, the performance for both strategies peaks exactly when the graph is equally partitioned.

In the case of LOW partitioning strategy, the resulting large GPU partition is vastly denser than those of the other two strategies. A denser graph leads to better locality, which the GPU is able to leverage efficiently because the associated "visited" bit-vector is small and fits the limited available cache (we confirmed this behavior via a set of experiments that we do not show for brevity). Hence, the GPU processing rate is much faster when using LOW compared to the other

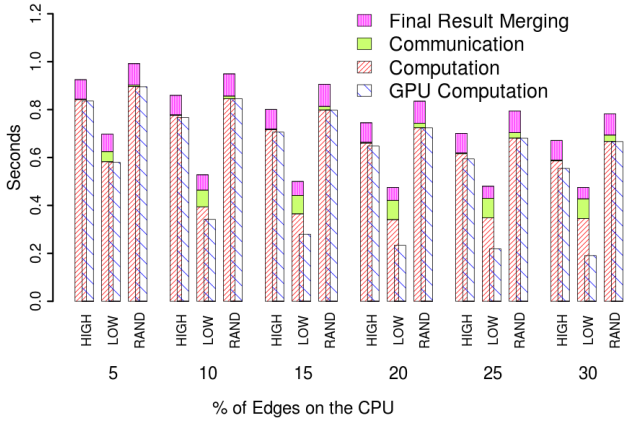

Figure 9: Breakdown of execution time for scale-25 graph. Note that the "Computation" bar refers to the computation overhead of the bottleneck processor. In the case where the GPU computation is the bottleneck, the GPU bar is as tall as the computation bar.

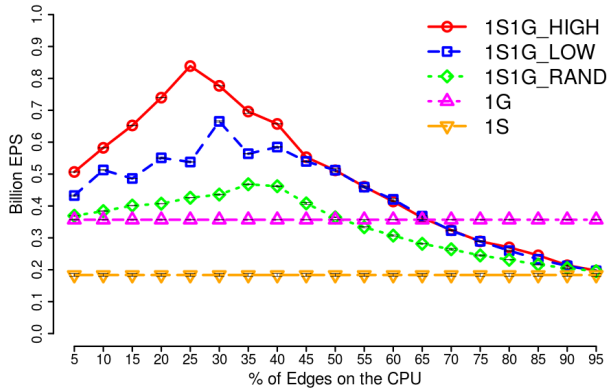

Figure 10: PageRank processing rate in edges per second for a scale 25 R-MAT graph.

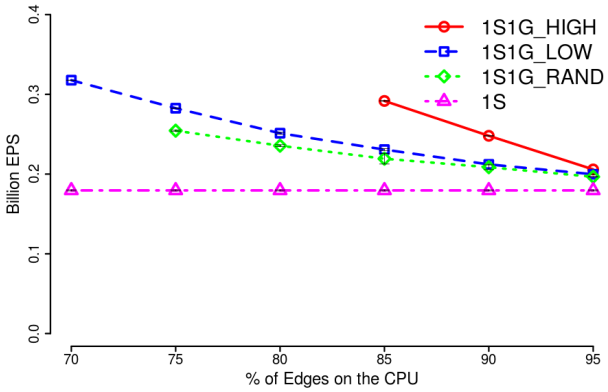

Figure 11: PageRank processing rate for a scale 28 R-MAT graph. Due to the small number of vertices assigned to the GPU partition, the LOW strategy is able to fit a larger percentage of the edges in the GPU.

two strategies for the same percentage of offloaded edges. In fact, the GPU performance is significantly more efficient to the degree that the bottleneck shifts to the CPU when increasing the CPU's share of edges to only 10%.

Summary so far. The previous two subsections provide explanations for the observed performance due to the graph partitioning strategies. Moreover, the results confirm that adding a GPU to a single CPU socket system is indeed a good opportunity to improve the efficiency of graph processing. While offloading some of the processing to the GPU is always beneficial, depending on the graph size, the partitioning strategy that delivers the best performance is different. In general, the rule for choosing a partitioning strategy is as follows:

- In the case of large graphs, when only a small fraction of the graph can fit in the GPU memory, HIGH partitioning offers the best performance as it allows the bottleneck processor, the CPU, to process its partition faster by creating a more cache-friendly partition.
- In the case of small graphs, when most of the graph fits in the GPU, the LOW strategy offers the best performance as it allows the GPU, now the bottleneck processor, to better utilize its abundant memory bandwidth by creating a GPU partition that is more memory coalesce-friendly.

C. *An Even Less Cache-Friendly Workload: PageRank*

In this section, we evaluate the PageRank algorithm. Compared to BFS, PageRank has a higher compute-to-memory access ratio, and does not employ summary data structures; hence the cache has a lower effect on processing performance on the host.

Figure 10 shows the processing rate in edges per second (EPS)[3] for a scale-25 RMAT graph. Similar to BFS, the performance increases until the workload is balanced between the two processors. Note that the GPU's processing rate is, again, higher than that of the host (each represented by a straight line in the figure); hence the peak point is shifted to the left for all strategies.

[3] To obtain a normalized performance metric, PageRank processing rate is computed by multiplying the total number of edges in the graph by the number of processing rounds and divided by processing time.

In the left end of the figure, HIGH performs the best among all partitioning strategies. Since PageRank does more computation per vertex than BFS, placing the few high-degree vertices on the CPU allows processing them faster as the CPU has cores clocked higher than the GPU ones. For the right end of the figure, the CPU is the bottleneck, and the performance improvement brought by the hybrid system is only proportional to the share of the graph offloaded.

Figure 11 shows PageRank processing for a larger, scale-28 R-MAT graph that does not entirely fit in the GPU. The behavior of the system is similar to that observed in the above discussed smaller graph.

Compared to the other two strategies, LOW partitioning allows for offloading a larger portion of the edges to the GPU (i.e., left of Figure 11). This happens because PageRank requires a larger per-vertex state than BFS; hence, the number of vertices assigned to a partition has a larger effect on a partition's memory footprint. Since LOW places the high degree vertices on the GPU, the number of vertices assigned to the GPU partition by LOW is lower than that assigned by HIGH and RANDOM strategies for the same number of edges.

As a result, although HIGH partitioning offers the best performance among the strategies when the number of edges offloaded is the same, LOW is able to fit more edges into the GPU, offering the best processing rate for this scenario.

D. *The Effect of Vertex Degree Distribution*

As discussed previously, most real-world graphs obey a power-law degree distribution. The fact that these graphs have skewed vertex degree distribution (*i*) guided our choice of partitioning strategies, and (*ii*) facilitated the aggregation optimization, which aims to reduce the communication overhead (where multiple edges from the same partition point to the high degree vertices, enabling aggregation).

To quantify the effect of vertex degree distribution on the above mentioned aspects, we evaluate the performance of the hybrid system using a case that is the worst input for TOTEM's optimizations: random graphs with uniform degree distribution. The graphs were generated using the Erdős–

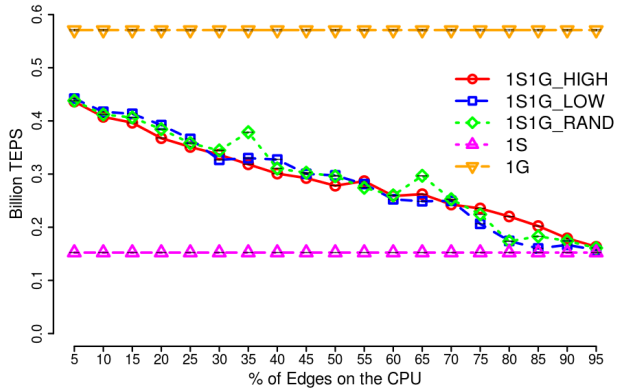

Figure 12: BFS traversal rate for a uniform scale 25 graph.

Rényi graph generation model [24]. The model generates edges with equal probability of setting an edge between any two vertices, independently of the other edges.

Figure 12 shows BFS traversal rate for a uniform scale-25 graph. A uniform graph fits best the GPU's SIMD processing model: the workload is balanced across the thousands of hardware threads, and allows more coalesced memory access patterns, hence utilizing better the high memory bandwidth of the GPU resulting in the GPU's superior performance compared to the CPU.

Figure 12 highlights that, when the graph has a uniform degree distribution the hybrid system performs almost the same irrespective of the partitioning strategy as all strategies produce partitions with similar characteristics. Because the GPU performs significantly better than the CPU, the hybrid system performance drops as the size of the CPU partition increases. Compared to processing the whole graph on the GPU, the hybrid system performance is inferior even when the majority of the graph is placed on the GPU (the left side of the figure). This is because the benefit from concurrently processing part of the graph on the CPU is masked by the communication overhead, which is more significant than for the R-MAT graphs.

Figure 13 shows the system's performance for the larger scale-28 uniform graph. Unlike the R-MAT workload and similar to the performance of the smaller graph above, all partitioning strategies perform similarly. Moreover, there is no gain from processing part of the graph on the GPU because the communication overhead masks the gain in

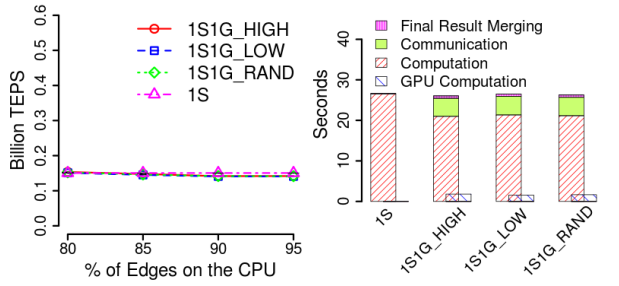

Figure 13: BFS performance on a scale-28 uniform graph. Left: traversal rate. Left: breakdown of execution time for the 80% data point.

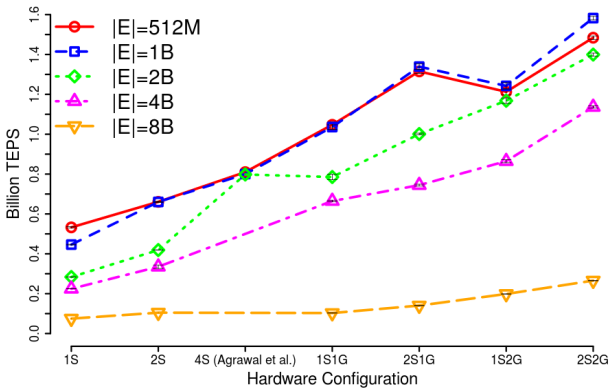

Figure 14: BFS traversal rate for different hardware configurations and different R-MAT graph sizes (scale-25 to 29). When GPUs are used the graph is partitioned to obtain best performance.

reducing the computation time, as the right figure shows.

E. Scalability: Adding More Processing Elements

This section focuses on the following questions: *What is more beneficial for graph processing, adding more CPU sockets or a GPU? How does the hybrid system scale with additional processing elements?* We believe that these are important questions because systems with two processing components (a CPU socket and a GPU) or one with few GPUs can become common-place due to their cost effectiveness.

Breath First Search. Figure 14 presents BFS traversal rate for different hardware configurations (up to four sockets/32 cores and 2 GPUs) and graph scales (scale-25 to 29).

First, we focus on the analysis of configurations with two processing units. The figure shows that for all graph sizes, the hybrid (1S1G) system performs faster than the dual-socket system (2S). Adding a second socket doubles the amount of last level cache, a critical resource for BFS performance, resulting in up to 1.5x speedup over a single socket system. However, the performance of 1S1G, brought by matching the heterogeneous graph workload with the hybrid system, outperforms that of the dual-socket symmetric system: between 1.57x to 2.2x speedup compared to the dual socket system.

Figure 14 also shows the performance of a quad-socket system (4S), as reported in the work by Agrawal et al. [1]. Each socket holds processors the same generation yet better than the ones we use: is a Xeon 7560, a Nehalem processor with 8 cores and 24MB LLC (hence 96MB LLC, 32 cores and 64 hardware threads on the system). Agrawal et al. reported BFS traversal performance on the smallest three graphs shown in Figure 14 (512M, 1B, and 2B). The abundance of LLC aggregated by the four socket system enables it to scale well with increased graph size. Still, the figure shows that a hybrid 1S1G system offers competitive performance to that of the four socket system (4S) at a lower cost in terms of both acquisition and running energy cost: using the Total Dissipated Power (TDP) values listed in Table 1, a quad-socket system requires 95*4=380W, while a

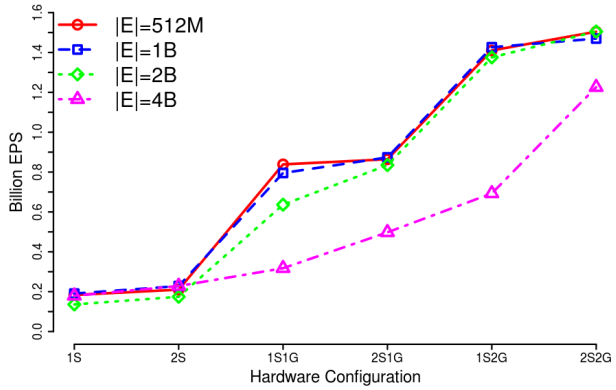

Figure 15: PageRank processing rate for different hardware configurations and R-MAT graph sizes (scale-25 to 28). Note that because PageRank requires more state per-vertex compared to BFS, we were not able to process a scale-29 graph like we did for BFS.

hybrid system requires 95+280=320W). In fact, 1S1G offers 1.29x speedup over the four socket system (4S) for the smallest two graphs.

The figure also shows the ability of the hybrid system to harness extra processing elements, achieving up to 1.6Billion TEPS for the smallest graph, and more importantly, as high as 1.13 Billion TEPS for a scale-28 RMAT graph (i.e., |E|=4B). It is worth pointing out that such performance is competitive, yet at a lower cost, with the performance results of the latest Graph500 list published as of writing this paper (June 2012) for graphs of the same size. Also note that TOTEM is a generic graph-processing engine, as opposed to the dedicated BFS implementations for most systems in Graph500.

Finally, Figure 14 shows the performance of a scale-29 graph. In general, the raw performance is much lower than the other smaller graphs due to the lower cache hit rate in the CPU partition (due, in turn, to a larger bit-vector). Moreover, handling such a large graph requires using 64-bit edge identifiers as the number of edges exceeds four billion, the maximum number of edge ids supported by a 32-bit type. This doubles the memory footprint of the graph, and significantly affects the percentage of edges that can be offloaded to the GPU, hence the raw performance of the hybrid systems are low compared to the other smaller graphs. Nonetheless, although a hybrid 1S1G system is able to offload only 1% of the edges to the GPU, it achieves 1.37x speedup compared to a single socket system, and similar performance to a 2S dual-socket one. Adding a second GPU increases the percentage of total offloaded edges to 5%, and the performance significantly improves: a 2S2G configuration achieves 2.5x speedup compared to a 2S dual-socket system. This highlights the vital importance of having more memory on the GPU for large graphs.

PageRank. Figure 15 shows a similar scalability plot for the PageRank algorithm. The figure shows that the CPU straggles in scaling when adding a second socket (1S vs 2S). Although PageRank does more computation per vertex than BFS, the incoherent memory access pattern combined with the inability of the algorithm to better utilize the available limited cache space renders the algorithm bound by main memory access latency; therefore adding more compute power via a second socket has little effect on performance.

Interestingly, adding a GPU sharply improves the performance. A hybrid 1S1G configuration achieves speedups between 1.5x and 4.5x compared to a 2S configuration. The benefit of offloading part of the workload to GPUs is confirmed again when adding a second a GPU where the plot shows another jump in performance for all workloads.

VI. RELATED WORK

Although graph algorithms are a well-studied area, leveraging parallel architecture to accelerate the algorithm runtime is far from a straightforward task. Parallel implementations require substantial effort to maximize the utilization of the parallel platform.

There is no shortage of work on graph partitioning for parallel processing. Traditionally, the problem is defined as to partition a graph in a balanced way, while minimizing the edge cut. It has been shown that this problem is NP-hard [12], therefore several heuristics were proposed to provide approximate solutions. We believe that classical solutions do not properly address the requirements for graph partitioning on hybrid platforms. Some heuristics such as Kernighan–Lin [21], has quadratic $O(n^2 \log n)$ time complexity, which is prohibitively expensive for the scale of the graphs we target. Multilevel partitioning techniques, such as METIS by Karypis et al. [19], offer an attractive moderate time complexity. In this case, partitioning is done by first coarsening the graph down to fewer vertices, resulting in a bisection of a much smaller graph, which is then computed (using one of the expensive techniques mentioned above). Finally, his smaller representation is projected back on the original graph.

However, as mentioned before, such techniques are mainly optimized to minimize communication, which is not a bottleneck in our case. Moreover, they target symmetric parallel platforms as they focus on producing balanced partitions, which is not sufficient for a hybrid system that has two processing units with largely different characteristics.

While we are unaware of previous works on optimizing graph processing on hybrid CPU and GPU systems, many efforts exist on optimizing graph algorithms on either the CPU or the GPU alone. For example, several studies focus on optimizing BFS on multi-socket CPUs [1, 8, 16]. One important recent effort by Chhugani et al. [8] applies a set of sophisticated techniques to improve the cache hit rate of the "visited" bit-vector, reduce inter-socket communication, and eliminate the overhead of atomic operations by using probabilistic bitmaps. Although our BFS kernel does not apply the aforementioned optimizations, our simplified BFS algorithm on one CPU socket and one GPU is 1.36x faster than Chhugani's algorithm on a comparable dual-socket CPU for a scale-28 RMAT graph. Our approach to partition the graph goes in the same direction in terms of improving

the cache hit rate on the CPU. We believe that their other techniques are complementary to our approach: they can be applied to the BFS kernel that runs on the CPU to improve the overall performance of the hybrid system.

Finally, past projects have explored GPU-only graph processing. However, these projects either assume that the graph fits GPU memory [15, 20], or they are tailored for multi-GPU setups [23]. In both cases, due to the limited memory space available on the GPU, the scale of the graphs that can be processed is significantly smaller than the graphs presented in this paper.

VII. LESSONS

The results presented in this work allow us to put forward a number of preliminary guidelines on the opportunity and the supporting techniques required to harness hybrid systems for graph processing problems. We stress that these guidelines are preliminary: they need to be hardened by experimenting with a larger set of algorithms, graphs topologies, and GPU models. We phrase these guidelines as answers to a number of questions.

- **Q:** *Does it make sense to use a hybrid system?*

  A: Yes, for scale-free graphs. One concern when considering using a hybrid system is the limited GPU memory that may render using a GPU ineffective when processing large graphs. We show, however, that it is possible to offload only a small portion of the graph to the GPU and obtain benefits that are higher than the proportion of the graph offloaded for GPU processing due to the heterogeneity of the graph workload. For instance, compared to a single socket system, BFS obtains 3x speedup on a scale-28 RMAT graph when offloading only 25% of the edges to the GPU, and almost 2x speedup compared to a dual-socket system (see Figure 14). For small graphs, although the GPU alone offers high processing rate, retaining part of the graph on the CPU boosts even more the GPU performance, and hence the hybrid system offers a 2x speedup on a scale-25 RMAT graph compared to a dual-socket system (see Figure 14).

- **Q:** *Is the partitioning strategy key for achieving high performance?*

  A: Yes, the low-cost partitioning strategies we explore – all informed by vertex connectivity – provide in all cases better performance than blind, random partitioning.

- **Q:** *Which strategies work best?*

  A: It depends. The answer is nuanced and the choice of the best partitioning strategy depends on the graph size and on the specific characteristics of the algorithm. If the graph is large, then the CPU will likely be the bottleneck as it is assigned the larger portion of the graph, where only a small fraction can be offloaded to the GPU. Thus, the goal of partitioning is to improve the CPU performance by producing and assigning to it the friendliest workload to its architecture. Our evaluation shows that placing the high degree vertices on the CPU offers the best overall performance: it improves the cache hit rate for algorithms that use summary data structures, and better balances the load across the cores for ones that do not use them.

  If the graph is small, most of the graph can fit in the GPU, and there is space to better balance the load between the two processing units. Since the GPU offers better processing rate, it will be assigned a larger portion of the graph. The policy for which partitioning strategy to use in this case is as follows. If the algorithm employs summary data structures, placing the low degree vertices on the CPU does not hurt its performance as the state is small enough to fit the cache, and the resultant GPU partition has a state small enough to fit its small cache, hence significantly improving its performance.

- ***Q:*** *Should one search for partitioning strategies that lead to higher performance by searching for partitioning solutions that reduce the communication overheads?*

  A: No. We show that, in the case of scale-free graphs, the communication overhead can be significantly reduced – to the point that it becomes negligible relative to the processing time – by simple aggregation techniques. Aggregation works well for four reasons. First, real-world graphs have skewed connectivity distribution. Second, the number of partitions the graph is split into is relatively low (only two for a hybrid system with one GPU). Third, aggregation can be applied to many practical graph algorithms, such as BFS, PageRank and Single-source Shortest Path. Fourth, there is practically no cost for aggregation: conceptually, aggregation moves the computation to where the data is, which must happen anyway. In contrast, partitioning algorithms that aim to reduce communication have typically high computational or space complexity and may be themselves 'harder' than the graph processing required [11].

- ***Q:*** *Is it possible to design a graph processing engine that is both generic and efficient?*

  A: Yes. A wide-range of graph algorithms can be implemented on top of TOTEM, which exposes similar BSP-based computational model and functionality to that offered by a number of other widely accepted generic graph processing engines designed for cluster environments (e.g., Pregel [22] and Giraph [29]). Our experiments show that being generic – that is, being able to support multiple algorithms and not only the popular Graph500 BFS benchmark, did not hinder TOTEM's ability to efficiently harness hybrid systems, and scale when increasing the number of processing elements. TOTEM's performance on a hybrid system with dual-socket and dual-GPU is capable of 1.13Billion breadth-first search traversed edges per second for a scale-28 RMAT graph, a performance that is competitive with (though at the bottom of) recent entries in the Graph500 list.

REFERENCES

[1] Agarwal, V. et al. 2010. Scalable Graph Exploration on Multicore Processors. *SuperComputing* (Nov. 2010).

[2] Ahn, Y.-Y. et al. 2007. Analysis of topological characteristics of huge online social networking services. *Proceedings of the 16th international conference on World Wide Web - WWW '07* (New York, New York, USA, May. 2007), 835.

[3] Barabasi, A.-L. 2003. Linked: How Everything Is Connected to Everything Else and What It Means. *Recherche*. 67, (2003).

[4] Barabási, A.-L. et al. 2000. Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: Statistical Mechanics and its Applications*. 281, 1-4 (Jun. 2000), 69–77.

[5] Chakrabarti, D. et al. 2004. R-MAT□: A Recursive Model for Graph Mining. *SDM* (2004).

[6] Chamberlain, B.L. Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations.

[7] Chambers, J.M. 1971. Algorithm 410 Partial sorting. *Communications of the ACM*. 14, 5 (May. 1971), 357–358.

[8] Chhugani, J. et al. 2012. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (May. 2012), 378–389.

[9] Chien, A.A. et al. 2011. 10x10: A General-purpose Architectural Approach to Heterogeneity and Energy Efficiency. *Procedia Computer Science*. 4, null (Jan. 2011), 1987–1996.

[10] Faloutsos, M. et al. 1999. On power-law relationships of the Internet topology. *ACM SIGCOMM Computer Communication Review*. 29, 4 (Oct. 1999), 251–262.

[11] Feldmann, A. 2012. Fast Balanced Partitioning Is Hard Even on Grids and Trees. *Mathematical Foundations of Computer Science 2012*. B. Rovan et al., eds. Springer Berlin / Heidelberg. 372–382.

[12] Garey, M.R. et al. 1974. Some simplified NP-complete problems. *Proceedings of the sixth annual ACM symposium on Theory of computing - STOC '74* (New York, New York, USA, Apr. 1974), 47–63.

[13] Gharaibeh, A. et al. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12* (New York, New York, USA, Sep. 2012), 345.

[14] Graph500: 2012. *http://www.graph500.org*.

[15] Hong, S. et al. 2011. Accelerating CUDA graph algorithms at maximum warp. *PPoPP* (New York, New York, USA, Feb. 2011).

[16] Hong, S. et al. 2011. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. *PACT* (Oct. 2011).

[17] Iori, G. et al. 2008. A network analysis of the Italian overnight money market. *Journal of Economic Dynamics and Control*. 32, 1 (Jan. 2008), 259–278.

[18] Jeong, H. et al. 2001. Lethality and centrality in protein networks. *Nature*. 411, 6833 (May. 2001), 41–2.

[19] Karypis, G. and Kumar, V. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*. 20, 1 (Dec. 1998).

[20] Katz, G.J. and Kider Jr, J.T. 2008. All-pairs shortest-paths for large graphs on the GPU. *SIGGRAPH/EUROGRAPHICS*. (Jun. 2008).

[21] Kernighan, B. 1970. *An Efficient Heuristic Procedure for Partitioning Graphs*.

[22] Malewicz, G. et al. 2010. Pregel: a system for large-scale graph processing. *SIGMOD* (Jun. 2010).

[23] Merrill, D. et al. 2012. Scalable GPU Graph Traversal. *PPoPP* (2012).

[24] P. Erdős, A.R. On the Evolution of Random Graphs.

[25] Seshadhri, C. et al. 2012. Community structure and scale-free collections of Erdős-Rényi graphs. *Physical review. E, Statistical, nonlinear, and soft matter physics*. 85, 5-2 (May. 2012), 056109.

[26] Steyvers, M. and Tenenbaum, J.B. 2005. The large-scale structure of semantic networks: statistical analyses and a model of semantic growth. *Cognitive science*. 29, 1 (Jan. 2005), 41–78.

[27] Valiant, L.G. 1990. A bridging model for parallel computation. *Communications of the ACM*. 33, 8 (Aug. 1990).

[28] Wang, X.F. and Chen, G. 2003. Complex networks: Small-world, scale-free and beyond. *IEEE Circuits and Systems Magazine*. 3, 1 (2003), 6–20.

[29] 2012. Apache Giraph.