A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing

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

Large, real-world graphs are famously difficult to process efficiently. Not only they have a large memory footprint but most graph processing algorithms entail memory access patterns with poor locality, data-dependent parallelism, and a low compute-to-memory access ratio. Additionally, most real-world graphs have a low diameter and a highly heterogeneous node degree distribution. Partitioning these graphs and simultaneously achieve access locality and load-balancing is difficult if not impossible.

This paper demonstrates the feasibility of graph processing on heterogeneous (i.e., including both CPUs and GPUs) platforms as a cost-effective approach towards addressing the graph processing challenges above. To this end, this work *(i)* presents and evaluates a performance model that estimates the achievable performance on heterogeneous platforms; *(ii)* introduces TOTEM – a processing engine based on the Bulk Synchronous Parallel (BSP) model that offers a convenient environment to simplify the implementation of graph algorithms on heterogeneous platforms; and, *(iii)* demonstrates TOTEM'S efficiency by implementing and evaluating two graph algorithms (PageRank and breadth-first search). TOTEM achieves speedups close to the model's prediction, and applies a number of optimizations that enable linear speedups with respect to the share of the graph offloaded for processing to accelerators.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel Programming*; D.3.4 [**Programming Languages**]: Processors – *Run-time Environments*; D.4.8 [**Operating Systems**]: Performance – *Measurements, Modeling and Prediction*; G.2.2 [**Discrete Mathematics**]: Graph Theory – *Graph Algorithms*.

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Heterogeneous Systems, GPU, Graph Algorithms, Breadth-first Search, PageRank, TOTEM.

1. INTRODUCTION

Graphs are the core data structure for problems that span a wide set of domains, from mining social networks, to genomics, to business and information analytics. In these domains, key to our ability to transform raw data into insights and actionable knowledge is the ability to process large graphs efficiently and at a reasonable cost. Imagine, for example, an advertising system for an online social network with hundreds of millions of users.

Many challenges make processing large graphs difficult. First, these workloads imply a large memory footprint (a "Mini" graph based on the Graph500 benchmark taxonomy [19] requires 137GB of memory). Second, most graph algorithms lead to a memory access pattern that has poor locality, data-dependent parallelism and a low compute–to-memory access ratio. Finally, most real-world graphs have a low-diameter and a highly heterogeneous node degree distribution (i.e., they are 'power-law') [2] thus partitioning them for access locality and load-balancing is difficult: processing graphs with low-diameter typically leads to a variable amount of parallelism available across execution rounds; while node degree heterogeneity causes imbalanced work across vertices.

Single-node[1] graph processing has a long history [1,12,14,17]. Yet, graph processing on single nodes is fundamentally limited not only by a node's memory size but also, and more importantly from a performance perspective, by memory access latency. Efficient single-node graph processing is, at the same time, a prerequisite to any efficient solution to process larger graphs on shared-nothing architectures (e.g., clusters).

This paper investigates the feasibility and the comparative advantages of supporting graph processing on heterogeneous, GPU-accelerated nodes. Moreover, in the spirit of building abstractions to hide complexity, it presents a generic graph-processing engine that leverages such platforms.

Two reasons make us believe that commodity heterogeneous nodes (e.g., GPU accelerated nodes) are an attractive building block to assemble a platform for high-performance, low-cost graph processing. First, GPUs bring massive hardware-supported multithreading able to hide memory access latency – a major performance bottleneck for this class of problems. Second, heterogeneous architectures that host processing units optimized for both fast sequential processing as well as for bulk processing match well the heterogeneous structure of graph workloads that need to be processed in practice. We expand on the opportunities and the challenges of using heterogeneous architectures in Section 2.

The following high-level questions guide our investigation:

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.

[1] We use *node* to refer to computing elements, and *vertex* to refer to the graph element.

Q1. *What are the general challenges to support graph processing on a single-node GPU-accelerated system?* In particular, *is it feasible to efficiently use both the traditional CPU cores and GPU(s) for graph processing*?

Q2. *Assuming that a low-level engine can efficiently process large graphs on GPU-accelerated nodes, what could an abstraction that aims to simplify the task of implementing graph algorithms look like*?

While we make preliminary progress on these relatively generic issues, this paper makes the following concrete contributions:

- We present a performance model that captures the key characteristics of a GPU-accelerated platform and the graph processing workload (Section 3). Given the characteristics of today's platforms (e.g., processing rates, memory space, and communication cost) the model demonstrates that *it is beneficial to offload part of the graph workload to a GPU*. Moreover, as we demonstrate, the model can serve as a tool to guide optimization efforts, system configuration, and provisioning.
- We present TOTEM, an open-source generic graph processing engine for GPU-accelerated platforms (Section 4)[2]. Our proposed design efficiently uses all CPU and GPU cores on a given node. Guided by the performance model, TOTEM applies a number of algorithm-agnostic optimizations that lead to performance improvements. One key optimization we introduce is reducing communication overhead by over an order of magnitude by aggregating messages at the source processor.
- We evaluate (Section 5) two key graph processing algorithms that are building blocks for a number of relevant applications: PageRank and breadth-first search (BFS). These two algorithms stand at the two extremes of a key performance factor for graph algorithms: the computation-to-communication ratio. We demonstrate linear speedups for both algorithms with respect to the proportion of the graph that is offloaded to a GPU.

The importance of this work comes from all these three contributions. Firstly, the performance model not only encourages the design of GPU-offloading techniques, but can guide hardware purchase and software design decisions for various classes of graph-related problems. Secondly, this work is the first to demonstrate the feasibility of using, in parallel, all CPU and GPU processors of a heterogeneous platform for a key class of irregular problems: graph processing. Finally, the processing engine we provide, TOTEM, offers an efficient and easy to use environment to develop graph applications that can benefit from acceleration.

Overall, our work is a basic building block towards an environment dedicated to processing graphs on large-scale machines (i.e., GPU-accelerated clusters, similar to the recent top entries in the Top500 supercomputer list). To this end, scaling out to multi-node systems is one of the main directions of future work (we elaborate on this in Section 6).

[2] The code can be found at: http://netsyslab.ece.ubc.ca

2. GRAPH PROCESSING ON HETEROGENEOUS ARCHITECTURES: OPPORTUNITY AND CHALLENGES

Large-scale graph processing faces two main difficulties. First, *large memory footprint:* efficient graph processing requires the whole graph to be present in memory, where for practical applications, graphs occupy from few gigabytes to terabytes of space. Second, *high memory access latency* combined with a *low computation to memory access ratio:* the major overhead of processing a graph is fetching the state of the vertices (e.g., a vertex iterates over its neighbors' state); because of poor locality and the scale of the workload, caches and prefetching are of little value, and most accesses are served by main memory.

The opportunity: GPU-acceleration has the potential to bring, at a low cost, the key advantage of massive, hardware-supported multithreading. In fact, a commodity GPU has much higher memory bandwidth and supports orders of magnitude more hardware threads and in-flight memory requests compared to traditional CPU processors. Additionally, properly mapping the graph-layout and the algorithmic tasks between the CPU(s) and the GPU(s) holds the promise to exercise each of these computing units where they perform best: CPUs for fast sequential processing (e.g., for the few high degree nodes of a power-law graph) and GPUs for the bulk parallel processing (e.g., for the many low-degree nodes).

Indeed, past work [5,8,10,12,16] demonstrates that GPU offloading offers tangible benefits compared to traditional multiprocessors for graph processing. However, *previous work assumes the entire graph fits in the GPU memory.* This is a major limitation as GPUs support up to one order of magnitude less memory space than the host (a trend that has been observed in the past ten years, and announcements of future GPU models indicate that this trend will continue at least in the medium term).

Hong et al. [8] work is, perhaps, the closest to this work in the spirit of enabling an application to harness the opportunities offered by a heterogeneous architecture, yet Hong et al. still assume that the GPU memory can hold the entire graph. In particular, they present a BFS implementation on a GPU-accelerated platform. They divide the processing into a first phase done on the CPU (as, at the beginning, only limited parallelism is available), and a second phase that starts once enough parallelism is exposed, at which point the whole graph is transferred to the GPU to accelerate processing.

The challenges: *This work investigates the feasibility of accelerating graph processing while partitioning the graph between the processing elements*. Moreover, in the spirit of building abstractions to hide complexity, it presents a graph-processing engine that leverages such platforms. To this end, we need to address a number of major challenges that we discuss in turn below:

First, the large amount of data to be processed, and the need to communicate between processors put pressure on two scarce resources: the GPUs' on-board memory and the host to GPU transfer bandwidth. Intelligent data placement solutions are needed to reduce memory pressure and limit generated transfer traffic. The ability to overlap communication with computation and to directly communicate between GPUs attached to the same host offer good prospects to mitigate some of these obstacles.

Second, to achieve good performance on GPUs, the application should match the SIMD computing model. As graph problems

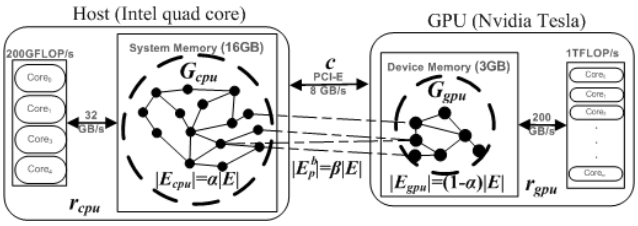

Figure 1: Graph partitioning on a heterogeneous node: An illustration of the model's parameters and their values for today's commodity components.

exhibit data-dependent parallelism, traditional implementations of graph algorithms lead to low memory access locality. Nevertheless, GPUs are able to hide memory access latency via massive hardware multithreading that, with careful design of the graph data structure, can reduce the impact of these factors.

Finally, mapping high-level abstractions (e.g., vertex-centric processing) and APIs to facilitate application development to the low level engine while limiting the efficiency loss, is an additional challenge.

Our methodology: To address these challenges, our methodology consists of the following interrelated steps:

- *Performance modeling*. We focus on assessing the feasibility of accelerating graph processing through partitioning and offloading a partition to the GPU. We take into account a number of key aspects such as the parallel processing model, the characteristics of the processing elements and the interconnect. It is worth noting that the model is agnostic to the precise graph algorithm and the model can be used to answer more complex resource provisioning and configuration questions.
- *Engine design and prototyping*. Informed by the performance model and the type of algorithms we aim to support, we design and prototype the building blocks of the graph processing engine (TOTEM). This includes defining the engine's programming model and interface.
- *Algorithm prototyping*. We prototype two graph algorithms on top of the engine. We evaluate the performance of the system to demonstrate the feasibility of graph partitioning and offloading to GPUs and validate and refine the performance model.

3. PERFORMANCE MODEL

The model aims to answer the following question: *Is it beneficial to partition and process the graph on both the host and the accelerator (when compared to processing only on the host)?*

In particular, we consider a heterogeneous node that consists of two processing elements $P = \{p_{cpu}, p_{acc}\}$, the CPU and an accelerator (e.g., a GPU) (Figure 1). The model can be easily generalized to a mix of multiple CPUs and accelerators. However, for the sake of simplicity, we use the setup with only two processing units for the rest of this section.

The model makes the following assumptions:

(i) Each processing element has its own local memory. The processing elements are connected by a bidirectional interconnect with communication rate c measured in edges per second (E/s) -- this is a reasonable unit as the time complexity of a large number of graph algorithms depends on the number of edges in the graph. However, the same model can be recast in terms of vertex-centric algorithms by normalizing by the number of vertices instead of edges.

(ii) The processing model is bulk synchronous parallel (BSP). Processing proceeds in rounds of concurrent computation and communication at the processing elements. Note that the performance model is agnostic to the architecture of each processing element.

(iii) p_{cpu} has enough memory to load and process the complete graph. The question is whether it is useful to partition the graph and process part of it on p_{acc}. p_{acc} has limited memory compared to p_{cpu}, and can not load and process the complete graph at once.

(iv) p_{acc} has higher graph processing rate than p_{cpu}. This is based on published results [8,12], which we validated independently.

(v) In addition to participating in graph processing, p_{cpu} schedules the workload (e.g., partitions the graph) and gathers the results. The model assumes these overheads are negligible compared to the algorithm processing time.

Let $G = (V, E)$ be a directed graph, where V is the set of vertices and E is the set of directed edges. $|V|$ and $|E|$ represent the number of vertices and edges, respectively.

The time it takes to process a partition of G, $G_p = (V_p, E_p) \subseteq G$ on a processing element p, is given by:

$$t(G_p) = \frac{|E_p^b|}{c} + \frac{|E_p|}{r_p} \quad (1)$$

where r_p is the processing rate of processor p (in E/s), and $E_p^b \subseteq E_p$ represents the subset of *boundary edges* – edges where either the source or the destination vertex is not located in p's local memory.

Equation 1 estimates the time required to process a partition as a combination of the time it takes to communicate possible updates through boundary edges (communication phase) plus the time it takes to process the edges in that given partition on processor p (computation phase). Intuitively, the higher the processing rate of a processing element, the lower is the processing time. Similarly, the less communication a processing element needs to access the edges in its partition, the lower is the processing time.

Now, we build on Equation 1 and define the makespan[3] of a graph workload G on a given platform P as follows:

$$m_P(G) = \max_{p \in P}\{t(G_p)\} \quad (2)$$

The intuition behind Equation 2 is that the performance of a parallel system is limited by its slowest component. Since, as discussed before, we assume that (*i*) the host processes its partition slower than the accelerator (i.e., $r_{cpu} < r_{acc}$) and that (*ii*) the host has more memory and is assigned a larger partition, the

[3] Makespan: the completion time of the last workload partition [13].

time spent on processing the CPU partition is always higher than that of the GPU partition (i.e., $t(G_{cpu}) > t(G_{gpu})$). Hence, the speedup of processing a graph on a heterogeneous platform (compared to processing it on the host only) can be calculated by Equation 3, as follows:

$$s_P(G) = \frac{m_{\{cpu\}}(G)}{m_P(G)} = \frac{m_{\{cpu\}}(G)}{m_{\{cpu\}}(G_{cpu})} = \\ = \frac{|E|/r_{cpu}}{|E^b_{cpu}|/c + |E_{cpu}|/r_{cpu}} \quad (3)$$

To understand the gains resulted from moving a portion of the graph to the accelerator, we rewrite Equation 3 by introducing two parameters that characterize the '*quality*' of the graph partition. Let α be the share of edges (out of the total number of graph edges $|E|$) assigned to the host, similarly let β be the percentage of *boundary* edges (i.e., the edges that cross the partition). Introducing these parameters, we have:

$$s_P(G) = \frac{|E|/r_{cpu}}{\beta|E|/c + \alpha|E|/r_{cpu}} = \frac{c}{\beta r_{cpu} + \alpha c} \\ = \frac{1}{\frac{\beta \cdot r_{cpu}}{c} + \alpha} \quad (4)$$

As expected, Equation 4 predicts that a high communication rate, c, improves the speedup. In fact, if c is set to infinity, the speedup can be approximated as $1/\alpha$. This is intuitive, as in this case the communication overhead becomes negligible compared to the time spent on processing the CPU's share of edges, α.

Figure 1 presents an illustration of the model and reasonable values for models' parameters for today's commodity heterogeneous platforms. We discuss them in turn next:

Communication rate (c) is directly proportional to the bus bandwidth and inversely proportional to the amount of data transferred per edge. The GPU is typically connected to the host via a PCI-E bus. The commonly used PCI Express 2.0 has a measured transfer bandwidth of 4GB/sec. If we assume the data transferred for each graph edge is a 4-byte value, the transfer rate c becomes 1 Billion E/s – or BE/s.

CPU's processing rate (r_{cpu}) depends on the CPU's characteristics, the graph algorithm, and the graph topology. We discuss possible values below in Table 1.

Percentage of boundary edges (β) depends on the graph partitioning between the processing elements. In the worst case all edges cross the partition.

The share of the graph that stays on the CPU (α) is configurable, but is constrained by the memory space available on the processing elements (for example, larger memory on the GPU allows for offloading a larger partition, hence smaller α).

We now explore the effect of changing the CPU processing rate or the percentage of boundary edges on the predicted speedup. Figure 2 shows the speedup predicted by the model (Equation 4) for different values of α, while varying the CPU processing rate (left plot) and the percentage of boundary edges (right plot).

Note that we keep at least half of the graph on the CPU (i.e., α*>50%)* as a conservative measure to ensure that the assumption

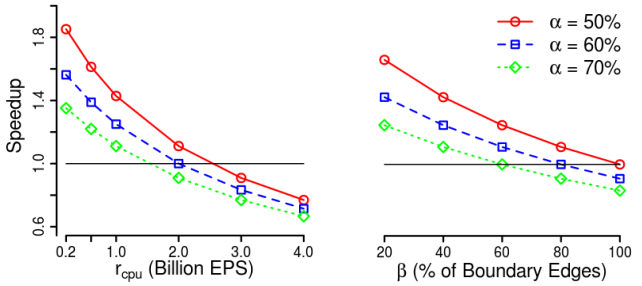

Figure 2: Predicted speedup (values below one indicate slowdown). Left: while varying the CPU's processing rate (β is set to 20%). Right: while varying the percentage of boundary edges (r_{cpu} is set to 0.5 BE/s). For both plots, the communication rate is 1 BE/s.

$t(G_{cpu}) > t(G_{gpu})$ (which allows us to transform from Equation 2 to Equation 3) always holds. Note that this constraints the speedup to 2x or lower.

Figure 2 shows that as r_{cpu} (left plot) or β (right plot) increase, the speedup decreases. For CPU processing rate above 1 BE/s or a graph partition that leads to larger or equal to 60% of boundary edges, the speedup decreases (with slowdown in some cases). This is because the communication overhead becomes significant. Figure 2 (right) also presents a hypothetical worst case where all of the edges are boundary edges (e.g., a bipartite graph where the partition cuts each edge). Even in this case, and due to the high communication rate c, the slowdown is not that significant, in fact, for α = 50%, no slowdown is predicted.

Figure 3 evaluates the effect of the amount of transferred data per edge on the predicted speedup. The speedup significantly drops as we double the amount of transferred data. However, having a low percentage of boundary edges, β, helps addressing this problem. Section 4.2 presents optimizations that aim to lower β (for practical cases we obtain an effective β as low as 3%).

To put in perspective the gains predicted by the model, and make sure we use values anchored in reality, we look at the estimated CPU processing rates based on the performance of the BFS algorithm benchmarks. Table 1 shows the best published parallel BFS performance results on a single CPU-only node for a power-law graph with 1Billion edges. BFS is a simple graph traversal algorithm with minimal per vertex processing. The processing rate for other more compute intensive algorithms, like Single Source Shortest Path (SSSP) and PageRank, is expected to be lower.

Nonetheless, as shown in the last column, offloading only 30% of the graph to a GPU offers tangible speedups (up to 1.39x).

More importantly, Table 1 allows us to get a sense of whether adding a GPU to the system brings higher benefits compared to adding extra CPU sockets.

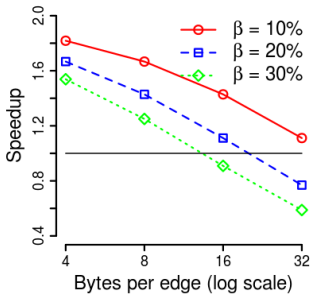

Figure 3: Predicted speedup while varying the transferred data per edge (α is set to 50% and the transferred bandwidth to 4GB/sec (PCI-E bandwidth)).

Table 1: Best published parallel BFS performance results (in ME/s) on a CPU-only node for an RMAT graph with $|V|$ = 32M and $|E|$=1B. The last column shows the performance and speedup predicted by the model if 30% of the workload is offloaded to a GPU (i.e., $\alpha = 70\%$). β is set to 3% (discussed in Section 4.2).

Ref.	Platform	Hardware Threads	r_{cpu} (ME/s)	Predicted rate with GPU (Speedup)
[1,8]	dual- Intel X5570	16	650	903 (1.39x)
	four- Intel X7500	32	1,050	1,435 (1.37x)

It is clear that the predicted speedup that results from adding a GPU to the dual-socket platform (i.e., 903 ME/s) leads to a performance that is comparable to the four-socket platform (i.e., 1050 ME/s), but at a lower price and energy point.

Considering the region of the plots where the parameters are set to values that represent realistic scenarios (650 ME/s ≤ r_{cpu} ≤ 1050 ME/s) we observe in Figure 2 that the model predicts speedups above one. Hence, we conclude that *it can be beneficial to have an engine to partition and process the graph on both the host and the accelerator*.

The model proposed here enables estimating the performance gained by replacing certain processing elements by accelerators (and vice-versa). As we have shown, there are practical scenarios where it is beneficial to add accelerators to graph processing platforms. The next challenge is to provide a framework that allows application programmers to write graph algorithms that make the most out of such heterogeneous platforms. We describe TOTEM, our proposal to address such challenge, in the next section.

4. TOTEM

To enable application programmers to leverage heterogeneous architectures, we design TOTEM – a graph processing engine for heterogeneous and multi-GPU single-node systems. This section presents TOTEM's programming model (Section 4.1) and implementation (Section 4.2).

4.1 Programming Model

TOTEM adopts a Bulk Synchronous Parallel (BSP) [15] parallel computation model and divides processing into rounds (*supersteps* in BSP terminology). Each superstep consists of three phases executed in order: *computation*, *communication* and *synchronization*. In the computation phase, each processor (in our case the CPU and the GPU(s)) executes asynchronously computations based on values stored in their local memories. In the communication phase, the processors exchange messages that are necessary to update their statuses before the next computation phase starts. The synchronization phase guarantees the delivery of the messages. Specifically, a message sent at superstep i is guaranteed to be available in the local memory of the destination processor only at superstep i+1.

Adopting the BSP model allows to circumvent the fact that the GPUs are connected via the high-latency PCI-Express bus. In particular, batch communication matches well BSP, and this enables TOTEM to hide (some of) the bus latency.

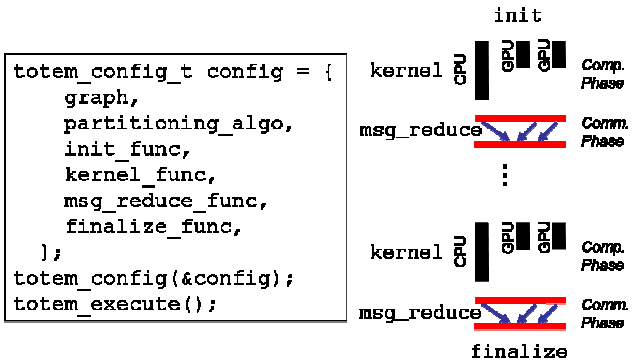

Figure 4: A simplified TOTEM configuration and how the callbacks map to the BSP execution model.

Computation phase. TOTEM initially partitions the graph and assigns each partition to a processing unit. In each compute phase of a superstep, the processing units work in parallel, each executing the user-specified kernel on the set of vertices that belong to its assigned partition.

Communication phase. TOTEM enables the partitions to communicate via boundary edges. The engine stores messages sent to remote vertices in local buffers that are transferred in the communication phase to the corresponding remote partitions. As the performance model shows, reducing communication overhead is paramount to improve performance. The engine achieves such reduction by aggregating at the source processor messages targeted to the same remote destination vertex. The aggregation is performed based on a user-provided function. Note that the synchronization phase is performed implicitly as part of the communication phase.

Termination. The engine terminates execution when all partitions vote to finish in the same superstep. At this point the engine invokes a user-specified function to collect and merge the results from all partitions.

4.2 Implementation

TOTEM is open-source, and is implemented in C and CUDA. A client configures TOTEM to execute a graph algorithm by implementing a number of callback functions executed at different points in the BSP execution model.

Figure 4 shows a simplified configuration of TOTEM. The engine creates one partition for the host and a partition for each GPU. The `init_func` allows the client to allocate algorithm-specific state (such as the cost array in BFS or the rank array in PageRank), the `kernel_func` callback performs the core computation of the algorithm, the `msg_reduce_func` callback defines how a message received from a boundary edge updates a vertex's state (e.g., update the vertex's state with the sum of the two in the case of PageRank, or the minimum in SSSP). Finally, the `finalize_func` callback enables the client to release state allocated at initialization. TOTEM accepts other configuration parameters, most notably is the graph partitioning algorithm the engine should use.

All callbacks are invoked per partition. If the partition is GPU resident, the engine ensures that the execution context is correctly set such that CUDA calls invoked from the callback are executed on the corresponding GPU.

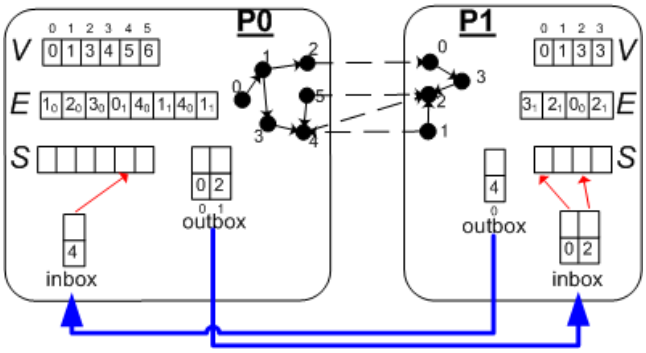

Figure 5: An illustration of the main data structures and the communication infrastructure in a two-way partitioning scenario.

A number of important aspects related to TOTEM's design and implementation are worth discussing; however, space constraints allow us to only discuss two topics: the data structures used and the communication via boundary edges.

Graph representation. Graph partitions are represented as Compressed Sparse Rows (CSR) in memory [3]**,** a space efficient graph representation that uses $O(|V| + |E|)$ space. Figure 5 shows an example of two-way partitioning setup. The arrays ***V*** and ***E*** represent the CSR data structure. In each partition, the vertex ids span a linear space from zero to $|V_p|$-1. A vertex id together with a partition id represents a global id of a vertex. A vertex accesses its edges by using its id as an index in ***V*** to fetch the start index of its neighbors in ***E***.

The array ***E*** stores the destination vertex of an edge, which includes the partition id (shown in the figure as subscripts) encoded in the high-order bits. In the case of boundary edges, the value stored in ***E*** is not the remote neighbor's id, rather it is an index to its entry in the outbox buffer (discussed later). To simplify state management, a vertex in a directed graph has access only to its outgoing edges, which is sufficient for most graph algorithms. Note that undirected edges can be represented in this graph data structure using two directed edges, one in each direction.

The array ***S*** represents the algorithm-specific local state of each vertex, it is of length $|V_p|$, and is indexed using vertex ids. Note that a similar array of length $|E_p|$ can be created if the state is required to be per edge rather than per vertex.

The processing of a vertex typically consists of iterating over its neighbors. A neighbor id is fetched from ***E***, and is used to access ***S*** for local neighbors, or the outbox buffer for the remote ones. Typically, accessing the state of a neighbor (either in ***S*** or in the outbox buffer) is done via atomic operations as multiple vertices may simultaneously try to update the state of a common neighbor.

To improve pre-fetching, the set of neighbors of each vertex in ***E*** are sorted and are placed such that the local edges are processed first (entails accessing ***S***), and then the boundary edges (entails accessing the outbox buffers).

Communication via boundary edges. A challenge for a graph processing engine for heterogeneous and multi-GPU setups is how to keep the cost of communication low. TOTEM addresses this problem by using local buffers and aggregation functions. As mentioned in section 4.1, messages sent via boundary edges in the computation phase of a superstep are temporarily stored and, if possible, aggregated in local buffers that are transferred in the communication phase.

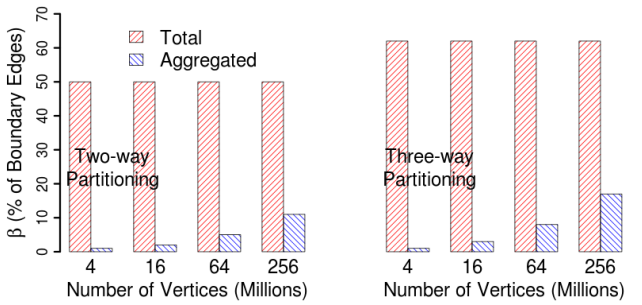

Figure 6: resulted *β* with and without aggregation while varying the number of vertices, and using a random partitioning algorithm. ($|E|$ = 512M).

TOTEM maintains two sets of buffers for each processing element (Figure 5). The outbox buffers have an entry for each remote neighbor, while the inbox buffers have an entry for each local vertex that is remote to another partition. An in/outbox buffer is composed of two arrays: one maintains the remote vertex id and the other stores the messages.

The outbox buffer in a partition is symmetric to an inbox buffer in another. Therefore, in the communication phase, only the message array is transferred. Once transferred, TOTEM uses the user-defined aggregation function to update the remote neighbors' state in the ***S*** array at the remote partition with the new values. Similar to ***E***, the entries in the inbox buffers are sorted by vertex ids to improve pre-fetching and cache efficiency when doing the update.

Optimizing access to boundary edges. To show the benefit of aggregating the communication along boundary edges, we test a naïve graph partitioning algorithm (i.e., random partitioning) and compare how much communication would happen with and without aggregation. Figure 6 shows β resulted from two- and three- way partitioning of a power-law graph (e.g., setup with one and two GPUs, respectively). The x-axis varies $|V|$ while keeping $|E|$ constant at 512M, hence varying the density of the graph (the density decreases as $|V|$ increases).

Increasing the number of partitions for the naïve partitioning increases β (from 50% for two partitions to 62% for three) as the probability of having a remote neighbor increases. More importantly, β is significantly reduced (between 3% to 16%) by aggregating boundary edges that have the same remote destination vertex. Note that decreasing the density of the graph reduces the opportunity for aggregation. However, even for a highly sparse graph, denoted by the most right data point in Figure 6, aggregation is still beneficial, reducing β by four times.

The worst case inputs are the Erdos-Renyi purely random graphs (The power-law RMAT graphs are advantageous as multiple edges from the same partition will point to the high degree vertexes and thus enable aggregation). Most graphs processed in practice, however, are power-law, thus this optimization is likely to be useful.

Finally, it is important to mention that aggregation works for algorithms where it is possible to reduce, at the source partition, into one value the values sent to the same remote vertex. For example, the "visited" status in BFS, minimum "distance" in SSSP and "rank" sum in PageRank.

Nonetheless, for algorithms where aggregation is not feasible, careful partitioning [9] has the potential to minimize the number of boundary edges, and reducing the communication overhead.

Summary of optimizations. In the following we summarize the main optimizations used by TOTEM. (they have been discovered through an iterative exploration process and provide sizeable gains. Lack of space prevents us to present a detailed evaluation of the performance impact of each of these optimizations).

(i) A compressed graph representation to reduce the memory footprint.

(ii) Aggregating boundary edges to reduce communication overhead. This technique has been used in cluster setups [11], and we show its application and effectiveness on heterogeneous platforms.

(iii) Sorting vertex ids in the inbox buffers to improve pre-fetching and cache efficiency when updating the vertices' local state.

(iv) Processing the local and remote edges separately to improve data access locality.

Limitations. Two main limitations related to the current implementation of TOTEM are worth mentioning. First, the implementation assumes fixed structure graphs, as the used CSR data structure makes it expensive to support updates to the graph structure during the algorithm execution (e.g., creation of new edges or vertices). This is a tradeoff. CSR enables a lower memory footprint and efficient iteration over the graph's elements (vertices and edges), which are essential for performance. Any other graph data structure that aims to enable mutable graphs will have to have some form of dynamic memory management (e.g., linked lists), which is expensive to support, particularly on GPUs.

The second limitation is related to the way communication is performed. During the communication phase of each superstep, the current implementation copies the whole outbox buffer of a partition to the inbox buffer of a remote partition assuming that there is a message to be sent via every edge between the two partitions. This is efficient for algorithms that communicate via each edge in every superstep, such as PageRank. However, this may be considered an overhead for other algorithms that communicate only via a selective set of edges in each superstep. For example, in the level-synchronized BFS algorithm, at a given superstep, only the vertices at the *frontier* communicate data via their outgoing edges. Additional compression techniques could be employed to lower the communication volumes.

Summary. The current implementation is space efficient, and simplifies the communication phase at the expense of adding communication overhead for some algorithms. An alternative implementation that enables selective communication via boundary edges, on the other hand, will avoid communicating inactive edges, but requires maintaining additional state (e.g., a bitmap to identify active remote edges or a thread-safe queue), and communicating a destination vertex id along with each message. We chose the former solution for its simplicity and acceptable performance, which is demonstrated in the next section.

5. EVALUATION

Goals. The evaluation aims to show the benefits of a heterogeneous platform compared to a platform based solely on CPUs. Particularly, it aims to address the following questions.

First, *how does TOTEM compare to the performance predicted by the model described in Section 3?* Such comparison allows us to validate the model and understand, for each use case, how much room is possibly left for optimizations.

Second, *how do the workload characteristics affect the performance predicted by the model and the one achieved by TOTEM?* We focus on one graph characteristic, graph density as it affects the communication overhead – a key overhead in distributed memory platforms.

Third, *how does TOTEM scale when increasing the number of GPUs?* New commodity systems are able to host multiple GPUs, which can be harnessed to offload larger part of the graph, and achieve better overall performance.

Workload. We use graphs generated using Recursive MATrix (R-MAT) scale-free graph generation algorithm [4], a graph generator adopted by the Graph500 benchmark. We use the benchmark's default configuration parameters (A=0.57, B=0.19, C=0.19 and D=0.05). In all experiments, we use a random partitioning algorithm. Finally, for each data point, we present the average over 20 runs.

Algorithms. We implemented two graph algorithms on top of TOTEM. We look at these algorithms as representatives of the two ends of the computation-to-communication ratio spectrum. First, PageRank, has a relatively high computation-to-communication ratio, and is less sensitive to communication overhead and memory access latency. The algorithm is based on the one described in [11]. It is worth noting that, to the best of our knowledge, this is the first work to implement and evaluate PageRank on GPUs.

Second, BFS, which has a low computation-to-communication ratio. BFS mainly does memory lookups with no major computation; hence it is more sensitive to memory access latency. The kernel implementation of BFS is based on the one described in [7].

Due to space constraints, we do not present the implementation of the algorithms. Also, all figures show two side-by-side plots for PageRank (left) and BFS (right).

Expected model prediction accuracy. We expect the model to predict better for PageRank than BFS. This is because PageRank is more computationally intensive; hence it will be less sensitive to the overheads introduced by TOTEM and not modeled (e.g., sub-optimal communication efficiency and extra memory lookups to handle boundary edges).

Testbed. We conduct the experiments on a machine with the following characteristics: dual-socket Intel Xeon (E5520) clocked at 2.27GHz per core, 16GB of host memory, two Tesla C2050 NVIDIA GPU (448 cores clocked at 1.1GHz, 3GB of memory). The cards are connected to the host via a PCI Express 2.0 x16 bus. The machine runs Fedora14, CUDA release 4.1 and driver version 64-285.05.33. TOTEM and the algorithms were compiled using g++ 4.5.1 with "-O3" option. OpenMP was used to parallelize the CPU code. Finally, all CPU-side processing was performed on the two CPU sockets.

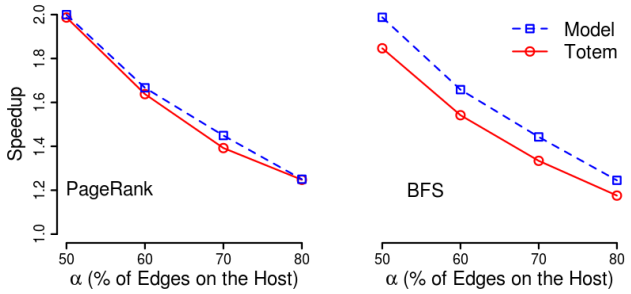

Figure 7: Predicted and achieved speedup while varying the percentage of edges on the host. (|*V*|=32M, |*E*|=1B).

5.1 TOTEM and the Performance Model

We first compare the speedup achieved by TOTEM and predicted by the model by offloading part of a graph to one GPU only. Figure 7 shows the speedup while varying α (|*V*| = 32M and |*E*| = 1B). The observed r_{cpu} is 155ME/s for BFS (which is comparable to latest numbers reported on a similar CPU model [8]), and 81ME/s for PageRank. *The observed β is 50%, but after aggregation, which both algorithms apply, β is reduced to only 2%.*

The plot shows that the achieved speedup follows the pattern predicted by the model. Also, as expected, the prediction is better for PageRank than for BFS.

Nonetheless, offloading 50% of the edges, which is the maximum we were able to fit on the GPU's 3GB device memory, offers tangible gains: out of the maximum possible 2x speedup indicated by the model, PageRank achieves 98%, while BFS achieves 84% of this maximum speedup.

To understand on which phase (computation or communication) and processor (CPU or GPU) the bulk of time is spent, we look at the breakdown of total execution time. Figure 8 shows the percentage of time spent on each phase. Two points are worth discussing.

First, the GPU processes its partition at a faster rate such that processing the CPU partition will remain the main bottleneck even when using a high-end CPU. For example, in the case of BFS, the GPU is 5 to 20 times faster. Recent published work on high-end dual-socket CPU, and using a highly tuned BFS kernel, (shown in Table 1) promises an at most 4x speedup compared to our measured CPU performance for the same workload. This

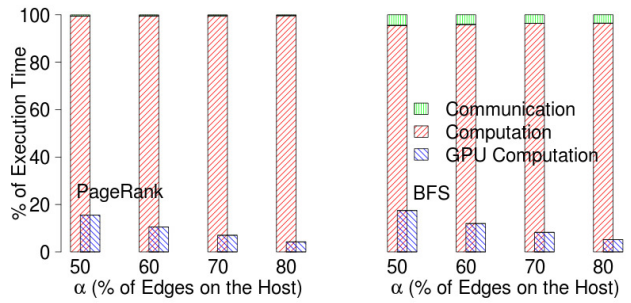

Figure 8: Breakdown of execution time. The computation phase is dominated by the processing of the CPU partition. The GPU partition, processed concurrently, is shown for comparison.

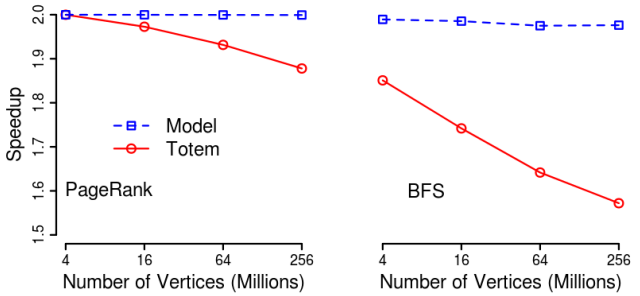

Figure 9: Predicted and achieved speedup while varying graph density. The number of vertices is indicated on the x-axis. (|*E*| = 512M). The higher the number of vertices, the lower is the graph's density. Note the compressed y-axis.

indicates that our assumption that the GPU finishes its processing faster will hold in practice.

Second, the CPU-GPU communication overhead is almost 20x lower than the computation. This is due to the aggressive aggregation of boundary edges which efficiently diminishes this overhead. Therefore, even when using a high-end CPU (and assuming 4x computation speedup), the computation will remain the main bottleneck.

5.2 The Effect of Workload Characteristics

To expose the effect of communication overhead, Figure 9 shows the predicted and achieved speedup while varying the density of the graph (by varying |*V*| and keeping |*E*| constant at 512M). Figure 6 (left) shows the observed β.

Figure 9 shows that the achieved speedup for both PageRank and BFS deviate from the model as the graph density is reduced. Figure 10 shows the predicted (based on Equation 1) and measured time spent on the communication and computation phases *for the data point where the deviation is the highest*, |*V*| = 256M.

In the case of BFS, the deviation in speedup is due to communication. As mentioned before, reducing the density of the graph while applying aggregation increases β; hence the communication overhead increases. Two factors generate this deviation. First, for the model, we set *c*, the communication throughput, to the PCI-E bus bandwidth. However, other practical factors affect the communication throughput, such as pre- and post-transfer overheads. TOTEM has a post-transfer overhead of

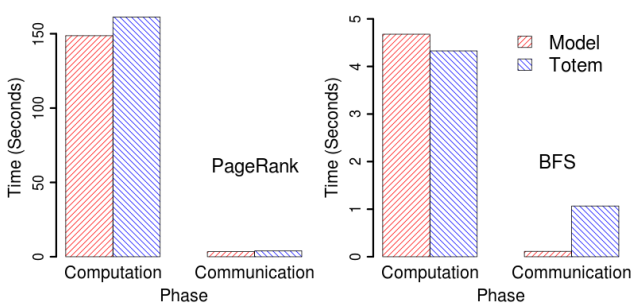

Figure 10: Predicted and measured time spent on the computation and communication phases for the datapoint where the deviation form the model is the highest (|*V*| = 256M, |*E*| = 512M).

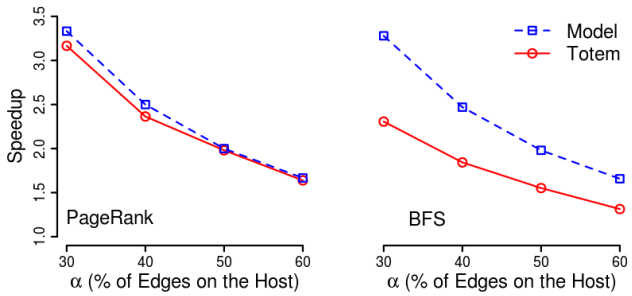

Figure 11: Predicted and achieved speedup after adding a second GPU. Note that having a second GPU allows using lower values of α as more edges can be offloaded. (|*V*|=32M, |*E*|=1.5B).

delivering the messages from the inbox buffer to the destination vertices' local state, which is not captured by the model. The second factor is related to a limitation in TOTEM's implementation (discussed at the end of Section 4.2) that results in extra transferred data, and almost an order of magnitude difference between predicted and measured communication.

In the case of PageRank, the computation phase dominates, and the small, less than 10%, speedup prediction error is generated by the achieved computation time which is sub-linear with respect to the offloaded proportion of the graph.

5.3 Adding a Second GPU

Adding a second GPU enables offloading a larger portion of the graph which, according to the model, provides additional speedup. Figure 11 shows the predicted and achieved speedup for a setup with two GPUs. In this experiment, we increased the number of edges of the graph to |*E*|=1.5B (|*V*|=32M). At α=30%, each GPU partition fills the GPU's device memory.

The figure shows that the model's prediction matches the achieved speedup for PageRank, while the memory-bound BFS deviates more from the model, yet it follows the trend indicated. As discussed in Section 4.2, increasing the number of partitions increases β, hence the communication overhead. Similar to the reason presented above, the deviation in the case of BFS is mainly the result of extra communication incurred by TOTEM's implementation. As α increases, less workload is offloaded, hence the number of boundary edges is reduced, rendering a lower communication overhead which results in better prediction accuracy.

Nevertheless, the results show that TOTEM is able to efficiently harness the added GPU, achieving linear speedup for PageRank and sublinear speedup for BFS.

6. SUMMARY AND DISCUSSION

GPU-acceleration is now a popular approach; however it has not been explored often for irregular applications. This paper investigates the feasibility of processing large graphs on single-node, GPU-accelerated heterogeneous platforms. In particular, the focus is on understanding whether (and in what scenarios) it is beneficial to harness GPUs to aid graph processing. In the following, we summarize the contributions of the paper and our plans to extend them.

Performance modeling. We presented a simple, yet effective performance model that helps estimating the benefits of offloading part of the graph workload. Given the current characteristics of heterogeneous platforms, the model shows that it is beneficial to partition the graph workload and process it on a heterogeneous platform, and highlights the importance of minimizing the communication overhead to improve the overall performance.

Notwithstanding the model's simplicity and its demonstrated usefulness, it can be improved (at the cost of making it more complex). For example, the model assumes that the processing rate r_{cpu} is constant, determined by a benchmark independent of the graph characteristics of the actual workload. A more accurate modeling of r_{cpu} would take into account the characteristics of the partition. To address this issue, we plan to analyze the effect of workload characteristics (e.g., degree distribution and graph structure) on obtained performance. We plan to perform controlled experiments on diverse hardware while varying graph characteristics, and feed the results to a machine learning approach to better predict r_{cpu} for partitions with specific characteristics.

Most importantly, considering the hardware characteristics as parameters in this machine learning methodology has the potential to predict what is more beneficial, adding more CPU sockets or accelerators, given a workload pattern and energy or dollar budget; hence providing valuable information needed for efficient system provisioning.

A graph processing engine for heterogeneous platforms. We presented the design and implementation of TOTEM, a graph processing engine for heterogeneous single-node platforms. We discussed a number of design decisions and optimizations TOTEM applies and their tradeoffs. TOTEM's importance, however, comes not only from enabling harnessing single-nodes, but also as a building block to harness GPU-accelerated clusters which have become common in the HPC space. For instance, three of the first five supercomputers in the latest (November, 2011) Top500 supercomputer list host GPUs [20] and heterogeneous architectures are increasingly popular [18].

Furthermore, TOTEM can be used as a back-end module of a domain specific language (DSL) for graph processing. For example, it can be used to extend the DSL developed by Hong et al. for graph analysis which currently targets only symmetric shared-memory platforms [6].

Our plan is to extend TOTEM to harness GPU-accelerated clusters. Shared-nothing architectures that aggregate heterogeneous nodes, that is, clusters of GPU-accelerated nodes, can offer a cost-efficient, yet high performance graph processing platform. The fact that new commodity nodes can support multi-hundred gigabytes of memory space, offers the opportunity to aggregate large memory space using smaller number of components; therefore, reducing the inter-node communication cost. At the same time, adding GPUs to each node offsets the loss in parallelism resulted from reducing the number of nodes.

Performance evaluation. We implemented and evaluated two graph algorithms on top of TOTEM. Our evaluation demonstrated the efficacy of the performance model, and the important gains offered by offloading part of the workload to one or more GPU.

We plan to extend our evaluation to more sophisticated graph partitioning algorithms than random partitioning. The ideal here is a low-cost partitioning technique that leads to partitions such that the generated workload for a partition matches well the strength of

the processing element the partition is allocated to (e.g., assign the few high-degree nodes of a power-law graph to fast sequential processor, the CPU, and the many nodes with a limited number of neighbors to the GPU).

Finally, we consider two future evaluation directions. First, investigating the tradeoffs offered by emerging heterogeneous architectures with shared memory between the CPU and the GPU, which eliminate the high-latency PCI-E communication bus (e.g., AMD Fusion). At a first look, the potential seems limited: as presented in the evaluation section, aggregation significantly reduces the communication overhead, eliminating it as a bottleneck. Moreover, since graph problems are memory-bound, integrated platforms have all processing elements compete to access the memory system, as opposed to platforms based on discrete GPUs. Still, such architectures can benefit from partitioning algorithms that map the graph partitions to the heterogeneous characteristics of the hardware. The second direction is to compare the energy and performance aspects of heterogeneous and symmetric architectures. Recent CPU models (e.g., Intel's Sandy Bridge) offer higher hardware multithreading and better memory bandwidth, two key improvements for graph processing. We believe that the performance model and its planned extensions will allow this type of exploration.

7. ACKNOWLEDGMENTS

We thank Greg Redekop and Samer Al-Kiswany for their insightful comments on earlier versions of this paper and for helping with early implementations of the algorithms we have used.

8. REFERENCES

[1] Agarwal, V., Petrini, F., Pasetto, D., and Bader, D.A. Scalable Graph Exploration on Multicore Processors. *SuperComputing*, (2010).

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

[3] Barrett, R., Berry, M., Chan, T.F., et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[4] Chakrabarti, D., Zhan, Y., and Faloutsos, C. R-MAT□: A Recursive Model for Graph Mining. *SDM*, (2004).

[5] Harish, P., Narayanan, P., Aluru, S., Parashar, M., Badrinath, R., and Prasanna, V. Accelerating Large Graph Algorithms on the GPU Using CUDA. *HiPC*, (2007).

[6] Hong, S., Chafi, H., Sedlar, E., and Olukotun, K. Green-Marl: A DSL for Easy and Efficient Graph Analysis. *ASPLOS*, (2012).

[7] Hong, S., Kim, S.K., Oguntebi, T., and Olukotun, K. Accelerating CUDA graph algorithms at maximum warp. *PPoPP*, (2011).

[8] Hong, S., Oguntebi, T., and Olukotun, K. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. *PACT*, (2011).

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

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

[11] Malewicz, G., Austern, M.H., Bik, A.J.., et al. Pregel: a system for large-scale graph processing. *SIGMOD*, (2010).

[12] Merrill, D., Michael, G., and Grimshaw, A. Scalable GPU Graph Traversal. *PPoPP*, (2012).

[13] Pinedo, M.L. *Scheduling: Theory, Algorithms, and Systems*. Springer Verlag, 2012.

[14] Scarpazza, D.P., Villa, O., and Petrini, F. Efficient Breadth-First Search on the Cell/BE Processor. *IEEE TPDS 19*, 10 (2008).

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

[16] Vineet, V. and Narayanan, P.J. CUDA cuts: Fast graph cuts on the GPU. *Conference on Computer Vision and Pattern Recognition Workshops*, IEEE (2008).

[17] Xia, Y. and Prasanna, V.K. Topologically Adaptive Parallel Breadth-First Search on Multicore Processors. *ICPDCS*, (2009).

[18] *TITAN: Paving the Way to Exascale*. 2011.

[19] Graph500. 2012. http://www.graph500.org.

[20] Top500. 2012. http://www.top500.org/.