Architectural Requirements for Energy Efficient Execution of Graph Analytics Applications
Invited Paper

Muhammet Mustafa Ozdal*, Serif Yesil†, Taemin Kim*, Andrey Ayupov*, Steven Burns*, and Ozcan Ozturk†
* {mustafa.ozdal, taemin.kim, andrey.ayupov, steven.m.burns}@intel.com
Intel Corp.
Hillsboro, OR 97124
† {serif.yesil, ozturk}@cs.bilkent.edu.tr
Bilkent Univ.
Ankara, Turkey

***Abstract*—Intelligent data analysis has become more important in the last decade especially because of the significant increase in the size and availability of data. In this paper, we focus on the common execution models and characteristics of iterative graph analytics applications. We show that the features that improve work efficiency can lead to significant overheads on existing systems. We identify the opportunities for custom hardware implementation, and outline the desired architectural features for energy efficient computation of graph analytics applications.**

I. Introduction

Especially in the last decade, we have seen enormous increase in the amount of data collected by companies and institutions. Machine learning applications have regained their popularity as people try to analyze and extract actionable information from this data. If the data is in the form of relations between individual entities, it can be represented as a graph, and graph analytics applications can be used to analyze it. Some examples are web graphs, social networks, and biological pathways.

Due to the high compute and storage requirements, graph analytics applications on big data are typically executed in data centers. It is common for large companies to customize their data centers based on the typical workloads they are running. With the energy and cooling costs dominating the data center operations, it becomes important to run these workloads in an energy efficient way without sacrificing performance.

In this paper, we focus on the characteristics of iterative graph-parallel applications that are hard to efficiently parallelize or accelerate using existing architectures. We limit our analysis to implementations that fit into the vertex-centric model as defined by distributed graph processing frameworks [1], [2].

Existing works on accelerating graph applications typically use a throughput metric such as number of vertices or edges processed per second. In this paper, we show that considering only throughput is not sufficient. A high throughput hardware or software implementation can be less work efficient, leading to longer execution times despite apparent performance gains. In this paper, we compare different execution models and study their effectiveness in terms of both throughput and convergence. We show that some features that improve convergence behavior can be inefficient to implement efficiently on existing architectures. We outline the architectural features that are needed for energy efficient computation of such applications on custom hardware implementations.

II. Application Characteristics

In this section, we describe some of the common characteristics of iterative graph parallel applications. Some example applications with these properties are PageRank, collaborative filtering, loopy belief propagation, and Gibbs sampling. In the rest of the paper, we will use PageRank as the driving example although most of the arguments also apply to other applications.

The PageRank algorithm is summarized in Figure 1 using the vertex centric model described in [2]. Here, r_u and d_u denote the page rank value and the out-degree of vertex u, respectively. In this implementation, the vertex program consists of 3 stages: Gather, Apply, and Scatter. In the Gather stage of vertex v, we compute the weighted sum of the page ranks of incoming neighbors[1]. Here, the weight of neighbor u is the number of outgoing edges from u. In the Apply stage, the new page rank of v (denoted as r_v^{new}) is computed by adding an offset value to the gathered sum. Here, α is a constant to ensure well-behaved iterations in the presence of vertices with no incoming or outgoing edges. In the Scatter stage, we check whether the rank of v has changed by more than the convergence threshold ϵ. If so, we schedule the outgoing neighbors of v because their ranks need to be recomputed to reflect this change.

Note that the vertex program is specified for a single vertex only. The actual implementation determines which vertices can be executed simultaneously under what conditions. We outline some of the execution characteristics in the next subsections, and we will discuss implementation options in Section III.

A. Asymmetric Convergence

The experiments in [2] show that the number of iterations each vertex needs to be processed varies significantly in typical

[1]Incoming neighbor of vertex v is a shorthand notation for another vertex u such that there is a directed edge $u \rightarrow v$.

```
Vertex program executed for each vertex $v$:
    $sum = 0$
    for each vertex $u$ for which $(u \to v) \in E$
        $sum = sum + \frac{r_u}{d_u}$
    $r_v^{new} = \frac{(1-\alpha)}{|V|} + \alpha \cdot sum$
    if $|r_v^{new} - r_v| > \epsilon$ then
        for each vertex $w$ for which $(v \to w) \in E$
            schedule $w$ for future execution
    $r_v = r_v^{new}$
```

Fig. 1. Vertex-centric representation of the PageRank algorithm

iterative graph algorithms. We have run the serial PageRank algorithm (Figure 1) on the LiveJournal benchmark [3] with weighting term $\alpha = 0.85$, and convergence threshold $\epsilon = 10^{-4}$. Figure 2 shows the cumulative histogram of the number of iterations each vertex is processed before it converges. According to this histogram, 7.4% of vertices converge in a single iteration, 50.8% converge in 36 iterations, and 99.7% converge in 50 iterations. On the other hand, only less than 0.3% of vertices need the full 77 iterations to converge. It is clear that it is not work efficient to process all vertices in every iteration.

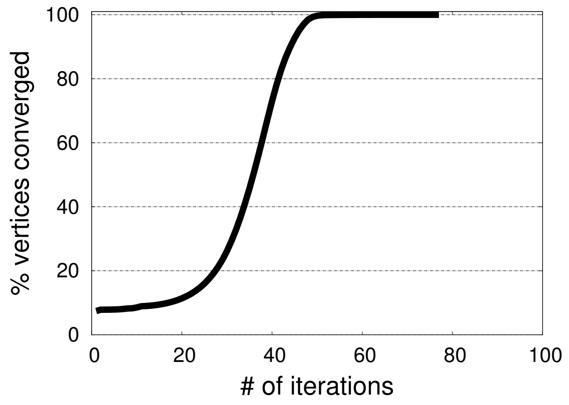

Fig. 2. The cumulative histogram of the number of iterations each vertex is processed for PageRank on the LiveJournal benchmark

It is a common practice in existing works to do performance comparisons using a throughput metric such as the number of vertices or edges processed per second. Such a metric does not properly penalize an implementation that processes all vertices in every iteration over an implementation that only processes the vertices that have not converged yet. Although the former is likely to have better throughput, the work efficiency of the latter might be much better, as will be shown later in this paper.

B. Synchronous vs. Asynchronous Execution

The Jacobi iteration formula for PageRank can be written as:

$$r_v^{k+1} = \frac{(1-\alpha)}{|V|} + \alpha \sum_{(u \to v) \in E} \frac{r_u^k}{d_u} \tag{1}$$

where r_u^k represents the page rank value computed for vertex u in the k^{th} iteration, and d_u is the out-degree of vertex u. Observe that only the rank values in the k^{th} iteration are used to compute the ranks in iteration $(k+1)$ in this formulation.

Alternatively, Gauss-Seidel iteration has the following formula:

$$r_v^{k+1} = \frac{(1-\alpha)}{|V|} + \alpha \sum_{\substack{u<v \\ (u \to v) \in E}} \frac{r_u^{k+1}}{d_u} + \sum_{\substack{u>v \\ (u \to v) \in E}} \frac{r_u^k}{d_u} \tag{2}$$

In words, when vertex u is processed in iteration $k+1$, it uses the rank values of iteration $k+1$ for the vertices before it and the rank values of iteration k for the ones after. It was shown in [4] that the Gauss-Seidel formulation can converge by about 2x faster than the Gauss-Jordan formulation in (1).

This concept can be generalized for other iterative graph problems [2]. If the data associated with a vertex computed in the current iteration is not used by other vertices in the same iteration, this is denoted as synchronous execution. Logically, all vertices are processed simultaneously, and they only need to access data from the previous iteration. In contrast, in the asynchronous mode of execution, when a vertex is processed, its data becomes available to other vertices in the same iteration. It has been shown that asynchronous execution converges much faster than synchronous execution for many graph applications [2].

Figure 3 compares the total work done for the PageRank algorithm in different modes of execution of the LiveJournal benchmark. As the x-axis, we use the total number of edges processed as proxy for the total work done. The y-axis is the number of vertices that have not converged to their final rank values. Here, "sync-all" refers to synchronous execution where all vertices are processed in every iteration. If the set of active vertices is maintained during synchronous iterations, it is denoted as "sync-active". Finally, the asynchronous mode of execution with active vertices is denoted as "async-active". Observe in Figure 3 that the least work-efficient mode is the synchronous execution where all vertices are processed in every iteration. Processing only the active vertices in synchronous mode reduces the total work done by almost 50%. Switching to the asynchronous mode reduces the work done by another 30%.

It is apparent that the *sync-all* mode is the easiest to implement in parallel. Since there are no intra-iteration dependencies, it is also likely to achieve the best throughput in terms of the number of vertices or edges processed per second. A common pitfall is to use throughput as the main performance metric, while ignoring the 2.75x increase in the total work done compared to the asynchronous mode of execution.

C. Data Access Bottlenecks

Typical graph applications perform small amount of computation per vertex or edge processed. For example, consider the PageRank algorithm in Figure 1. For vertex v, we need to add up the weighted rank value of each vertex u for which a directed edge $(u \to v)$ exists. In an unstructured graph, it

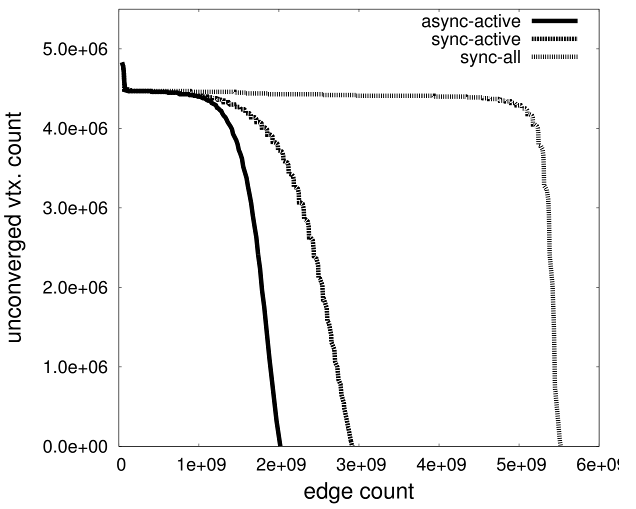

Fig. 3. Convergence behavior of PageRank on the LiveJournal benchmark

is hard to ensure that the data associated with neighboring vertices are stored in nearby memory locations. Especially for large graphs that do not fit into cache, each access to neighbor vertex data may require a transfer from main memory, which may take more than 100 processor cycles. Obviously data access costs dominate the computation costs in such cases.

D. Power Law Distribution of Vertex Degrees

It has been observed that the vertex degrees of real world graphs follow the Power law distribution [2], [5], [6]. In other words, a small percent of the vertices are connected to most of the edges. For example, in a social network, there can be celebrities with millions of followers, while majority of the users have at most tens or hundreds of followers. In a parallel implementation, vertex-based partitioning can lead to load imbalances as will be discussed further in Section III-D.

III. Implementation Challenges and Customization Opportunities

In this section, we will discuss the challenges of implementing graph-parallel applications that have the execution patterns discussed in Section II. We will also outline the architecture requirements for custom hardware implementations targeted for such applications.

A. Active Vertex Set

As discussed in Section II-A, it is important to maintain the set of *active* vertices when different vertices require different number of iterations to converge. In general, when the data associated with a vertex is modified significantly, its neighbors might need to be processed (again) to reflect this change in their data.

In a serial implementation, this can be achieved by maintaining an efficient set data structure that prevents multiple vertex copies at a given time. When multiple vertex programs are executed in parallel, implementation of such a set can be more complicated because multiple threads might try to write to it simultaneously causing race conditions. This requires atomic access support, and can lead to some performance loss due to serialization.

Alternatively, a relatively simple implementation is to keep a predicate per vertex indicating whether a vertex needs to be processed or not. Note that in the synchronous mode, there are two copies of data and control objects: for the current and next iteration. So, multiple threads can activate a vertex v without serialization, because they simply need to set v's predicate for the next iteration to 1. When vertex v is processed, its predicate for the current iteration is set to 0. This allows race-free execution while avoiding extra locking overheads. On the other hand, this approach would not work for the asynchronous mode of execution, because only a single copy per data and control object is maintained. It is possible that multiple threads try to set the same predicate $0 \rightarrow 1$ and $1 \rightarrow 0$ simultaneously. Furthermore, this approach has an extra overhead because the whole predicate array needs to be scanned to find active vertices. In the example of Figure 2, only 0.3% of the vertices are active in the last 27 (out of 77) iterations, and the whole predicate array needs to be scanned to find the active vertices.

SIMD architectures such as GPUs also have control divergence issues. Simultaneously executing threads (e.g. a warp in CUDA) can have both active and inactive vertices assigned to different threads, which might lead to underutilized hardware resources.

Ideally, a custom hardware implementation needs to implement special mechanisms to allow high throughput multiple simultaneous updates to the active vertex set. It needs to be stored in main memory for large graphs with several millions of vertices. On the other hand, it should allow efficient caching/buffering mechanisms to allow high-throughput and low-latency access.

B. Asynchronous Execution Support

It was shown in Section II-B that asynchronous execution can be significantly more work efficient than synchronous. The original algorithms (e.g. Gauss-Seidel iterations) assume that the execution of vertices happens in sequential order. However, parallel execution of vertex programs does not necessarily correspond to a sequential order, which may lead to correctness or convergence issues for some applications.

As a straightforward example, consider a simple graph coloring implementation, where we assign a color for vertex v that is different from the colors of v's neighbors. Although not optimal, serial execution of such an algorithm can find a solution as long as the number of colors is large enough. On the other hand, parallel execution is not *guaranteed* to find a feasible solution. For example, consider two vertices u and v with an edge between them. If u and v are always executed simultaneously, they might end up having the same color every time they are updated because they only see the previous color of each other.

The issue here is related to the *sequential consistency property*, which is the condition defined as *"the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order* *specified by its program"*. [7]. In the example above, there is no guarantee that the parallel execution will correspond to a sequential order logically. Some algorithms converge faster with sequential consistency (e.g. Alternating Least Squares), and some rely on it for correctness (e.g. Gibbs sampling) [2].

Sequential consistency is not easy to enforce when multiple vertices are processed in parallel. For typical GPU systems, locking mechanisms are limited, and may incur significant performance overheads. CPUs have better locking mechanisms in general, but the overhead of implementing sequential consistency can be significant. For example, the GraphLab engine [2] slows down by at least an order of magnitude on a shared-memory many-core system for PageRank when sequential consistency is enabled (see Section IV).

For custom hardware, special synchronization mechanisms to ensure sequential consistency can be implemented with low performance overhead. Conceptually, the idea can be similar to out-of-order microprocessors, where dependencies between different vertex programs are determined dynamically.

C. Latency Tolerance

An out-of-order (OOO) core is capable of reordering instructions to hide the long latencies of memory accesses. However, the state-of-the-art general purpose CPU architectures are designed assuming that most applications have reasonable levels of access locality, and they rely on caches to hide the long latencies of memory accesses. For many graph applications, this may not be the case especially if the graphs are unstructured and their sizes are much larger than the available caches.

For an OOO core, one of the limitations is the number of available line fill buffers (LFBs), which are used to service cache misses. For example, a single Haswell core has 10 LFBs, limiting the number of outstanding memory requests to 10 per core. For this example, let us consider the case where DRAM latency is 70ns and bandwidth is 64GB/s with access granularity of 64 bytes. To hide DRAM access latencies while utilizing the full DRAM bandwidth, we need to have 70 outstanding memory requests, which is 7x larger than what a single core can provide.

In other words, a state-of-the-art general purpose core underutilizes the available DRAM bandwidth for such applications, and needs to stall when there is not enough computation per data object accessed from memory, as is typically the case for graph applications (see Section II-C). Using multiple cores can allow full utilization of the DRAM bandwidth, but this reduces the energy efficiency by increasing the number of stalled cores.

A custom architecture can be designed that can saturate DRAM bandwidth utilization by issuing many independent memory requests corresponding to different vertices and/or edges. Dependencies between these requests need to be respected especially if sequential consistency is required.

D. Dynamic Load Balancing

Consider a parallel implementation where vertices of a graph is assigned to different threads. Due to power-law distribution of vertex degrees, it is possible that some threads need to process significantly more edges than the others. Software based dynamic load balancing techniques typically can address this issue effectively for multi-core processors. However, this may cause performance issues for straight-forward GPU implementations with static vertex-to-thread mapping especially due to control divergence. For this reason, edge-based GPU implementations are used for applications such as stochastic gradient descent [8] and single source shortest path [9]. However, edge-based implementations have other issues when multiple edges of the same vertex are trying to update the same vertex data.

Dynamic load balancing should be considered together with latency tolerance techniques for custom architectures. Latency tolerance may require many independent light-weight threads running on the custom hardware. The architecture needs to be efficient even when the vertex degrees vary significantly.

E. Access Pattern Customization

There are different types of objects that are accessed when a vertex program is executed. In a typical sparse graph representation, indices of the edges connected to a vertex are stored contiguously (edge-list), while the offsets to this array are stored in a separate array per vertex (vertex-list). In addition to the graph topology, there can be additional arrays to store vertex and edge data. The access patterns to different types of data structures should be studied for the target class of graph applications. For example, it is typically the case that accesses to edge-list have good spatial locality, because all edges connected to a vertex are likely to be processed one after another. On the other hand, the access locality of vertex or edge data arrays can be poor due to the random nature of accesses.

If a specific hardware accelerator is to be designed for certain class of graph applications, the memory subsystem should be customized for high performance and energy efficiency based on the data access patterns.

IV. Experimental Results

In this section, we analyze 3 different PageRank implementations in the GraphLab framework on a server with 24 IvyBridge cores and 128GB memory. We use GraphLab version 2.2 in our experiments, and use different options to run PageRank in different modes:

1) *Sync*: The synchronous execution mode described in Section II-B, where the updated data for a vertex or edge becomes available in the next iteration for the neighbors. Iterations are separated by barriers. An active set is maintained so that only the vertices that have not converged yet are processed.
2) *Async-FC*: The asynchronous execution mode described in Section II-B, where the data updated for a vertex or edge immediately becomes available for the neighbors. Iterations are not well defined, and no barrier is needed. Sequential consistency is not guaranteed. Instead, a weaker consistency model called *factorized consistency (FC)* is used to ensure data consistency per edge only [2]. Similar to *Sync*, only the unconverged vertices are processed.

3) *Async-SC*: Similar to *Async-FC*, except that sequential consistency (SC) is guaranteed by locking neighboring vertices before starting to process each vertex.

We use 3 real-life benchmarks: Pokec ($|V| = 1.6M$, $|E| = 31M$), Web-Google ($|V| = 0.9M$, $|E| = 5.1M$), and Live-Journal ($|V| = 4.8M$, $|E| = 69M$), where $|V|$ and $|E|$ represent the number of vertices and edges, respectively.

In our first experiment, we analyze the work efficiency based on the discussion in Section II-B. For PageRank, we define the total work done as the total number of edges processed across all iterations. As can be observed from Figure 1, the main bottleneck is the inner loop where the weighted sum of neighbors is computed. Figure 4 compares the work efficiency of the 3 execution modes we have studied, where *async-FC* and *async-SC* values are normalized with respect to the *sync* value for each benchmark, and lower bars correspond to higher work efficiency. As can be observed from this chart, the asynchronous mode of execution needs to process about 40% less number of edges for convergence compared to the synchronous mode. This chart also shows that enabling sequential consistency does not noticeably change the convergence characteristics of PageRank application.

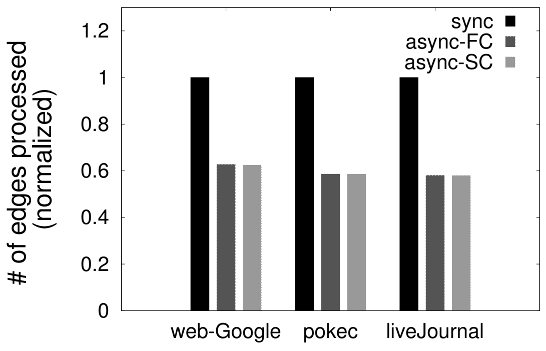

Fig. 4. Number of edges processed

In the second experiment, we analyze the throughput in terms of the number of edges processed per second. Figure 5 shows the throughput values normalized with respect to the *sync* mode. Observe that the throughput of *async-FC* mode can be up to 40% lower depending on the graph topology. Furthermore, the throughput of *async-SC* mode is more than 95% lower because of the overhead of fine-grain locking.

The first two experiments have shown that the synchronous and asynchronous modes have different advantages in terms of work efficiency and throughput. In the third experiment, we compare the total runtimes to understand the combined effect of these factors. In this experiment, we do not include the results of *async-SC*, because they are 20-30x larger than the others, as indicated by the low throughput values in Figure 5. Interestingly, for 2 out of 3 graphs, the runtimes of *sync* and *async-FC* modes are very similar. This indicates that the work efficiency advantage of the asynchronous mode is canceled out by the overheads associated with implementation. For the Web-Google graph, we observe better runtime for *async-FC*, mainly because of similar throughputs.

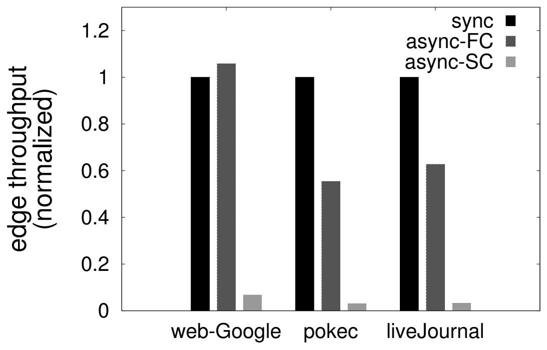

Fig. 5. Edge throughput

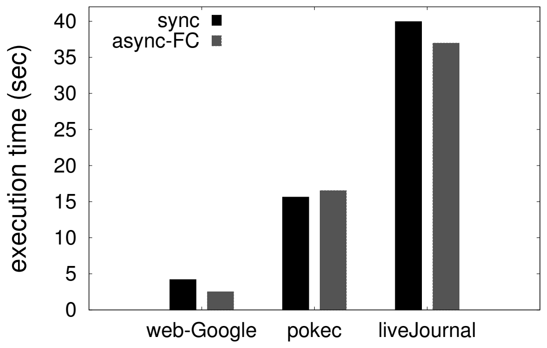

Fig. 6. Execution time

These experiments show that asynchronous mode of execution has the potential to improve performance and energy efficiency significantly for iterative graph applications. However, extra overheads associated with existing platforms may prevent achieving this benefit. This can be addressed by custom architectures targeted at these types of applications.

V. Conclusions

In this paper, we have outlined the common characteristics of iterative graph analytics applications. We have discussed the limitations of and overheads associated with existing multi core and GPU systems. We have also summarized the architecture requirements for custom accelerators to address these issues for better performance and energy efficiency.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[2] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, pp. 716–727, Apr. 2012.

[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, (New York, NY, USA), pp. 44–54, ACM, 2006.

[4] A. Arasu, J. Novak, J. Tomlin, and J. Tomlin, “Pagerank computation and the structure of the web: Experiments and algorithms,” 2002.

[5] A.-L. Barabasi and R. Albert, “Emergence of scaling in random networks,” *Science*, vol. 286, pp. 509–512, October 1999.

[6] S. Suri and S. Vassilvitskii, “Counting triangles and the curse of the last reducer,” in *Proc. of the 20th Int’l Conf. on World Wide Web (WWW)*, pp. 607–614, 2011.

[7] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Trans. Comput.*, vol. 28, pp. 690–691, Sept. 1979.

[8] R. Kaleem, S. Pai, and K. Pingali, “Stochastic gradient descent on gpus,” in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, (New York, NY, USA), pp. 81–89, ACM, 2015.

[9] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on gpus,” in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pp. 141–151, Nov 2012.