On the Design of an Efficient Hardware Accelerator for Large Scale Graph Analytics

Y.S.Horawalavithana
University of South Florida
Tampa, Florida
Email: sameera1@mail.usf.edu

***Abstract*—Graph analytic applications have gained traction as an expressive alternative to mine rich insights but are often suffered from memory latency and bandwidth bound issues over general purpose computing. In our study, we focus on designing a hardware accelerator to improve the efficiency of large scale graph processing while mitigating these problems. We will review existing hardware accelerator approaches for graph computation that exhibit specialized computation patterns including irregular memory accesses, iterative processing, and burst workloads. We will discuss key design choices on designing such approaches to gain advantage over graph execution characteristics. Further, a comparison over different approaches will be provided with experimental results.**

***Index Terms*—graph analytics, accelerators**

I. Introduction

Graph processing has been emerged as a new computation paradigm over large scale data analysis. There are many real world problems that can be solved using graph analytics which produce new insights, ranging from modern web-search to breast cancer treatments. Existence of many graph processing frameworks would prove the recent interest over such practices. However, due to the random nature of graph modeling and computation, many former software frameworks are limited to the support that general purpose processors could provide [1].

We focus on two popular graph computation patterns in our study; vertex-centric [2] and gather-apply-scatter (GAS) [3]. In vertex-centric graph computation, it's expected that a chain of frequent memory accesses would occur due to neighbor traversals as shown in Figure 1.

In gather-apply-scatter (GAS) memory model, the nodes collect information about 1-hop neighbors and integrate them to update node states. They are redistributed among other nodes after specific states have been obtained (Figure 2).

On the contrary, both these memory models are irregular over random neighborhoods which lead graph computation to be frequently suffered from poor cache locality on top of general purpose processors. Thus, on-chip memory is not effectively utilized due to randomness of graph specific data-types, and also off-chip memory bandwidth is wasted due to out of pattern access [1]. Also it could yield a significant delay to get data from memory to computation units. Hence, subsequent computation may require longer time in the pipeline since many graph traversals are memory intensive.

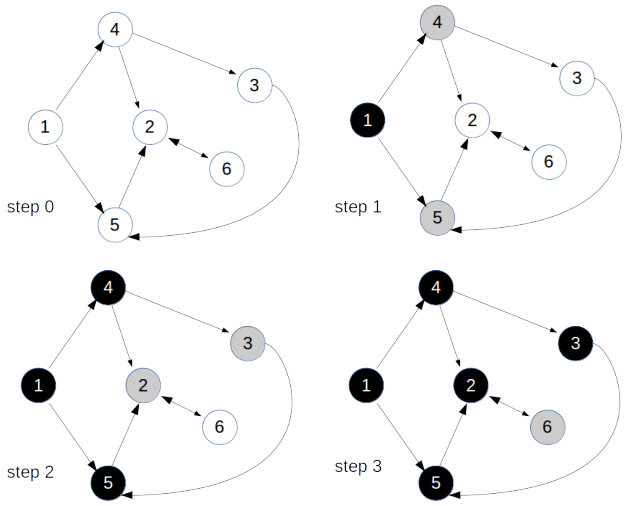

Fig. 1. Vertex-centric memory model

Memory models have been scaled out to mitigate such issues. But it could provide more additional complexity to the system when increasing the number of servers to gain larger in-memory storage, and would be a less viable option in cost. Moreover, burst graph workloads would be more decisive at the bottleneck. In contrast, there exists different implementation choices such that it even becomes harder to improve concurrency on top of traditional processors. It's noticed that general purpose instruction set is not tailor made for the domain of graph applications [1].

In other hand, one could exploit on improving underlying hardware architecture to improve the efficiency of off-the-shelf in-memory graph computation. As a result, hardware acceleration has attracted a lot of attention recently as an alternative to improve the execution of operations in specific data-structures[1]. Hardware accelerators could be designed as a separate unit from CPU for memory intensive graph algorithms in large scale processing [4]. In this paper, we focus on designing such accelerator architectures. Accelerators have been considered as specialized memory systems that do cater

[1]https://en.wikipedia.org/wiki/Hardware_acceleration

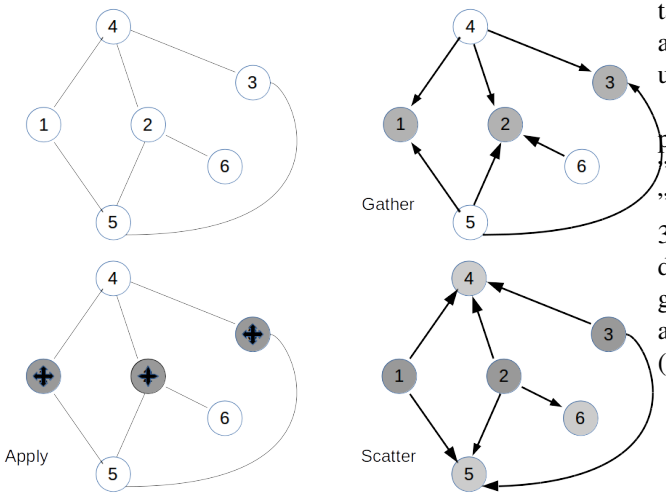

Fig. 2. Gather-apply-scatter memory model

efficient on-chip memory usage as well as an inherent load balancer to effectively manage burst graph work-loads.

We identify following objectives on designing an efficient accelerator architecture for large scale graph processing;

- Scale system performance with memory bandwidth
  - To reduce latency of moving data between computation units and memory
  - To process iterative computation steps faster
- Utilize hardware resources more efficiently to save computation energy

We would limit our study to hardware accelerators which would focus on improving efficiency on cores and in-memory computation in the domain of graph processing.

The rest of our study is structured as follows. In Section II, we explain recent hardware accelerator models exploited in graph domain. Section III discuss the key challenges found at the implementation of such models. Also we will outline a comprehensive comparison in the summary. Section IV summarize the study and outline the future work.

II. Hardware Accelerator Models

In this section, we review accelerator models over the domain of FPGA, GPU and 3-D stacking, outlining design decisions, data-structures, optimization strategies, evaluation mechanisms and experimental results.

A. FPGA

Field Programmable Gate Arrays (FPGA) is a custom accelerator model which provides the flexibility for application-specific programmers to utilize a number of logical gates and DRAM blocks in a FPGA board via a hardware domain language[2].

[2]https://en.wikipedia.org/wiki/Field-programmable_gate_array

1) FPGP [5]: FPGP is an on-chip parallel processor designed on top of FPGA to cater vertex-centric graph computation. The approach is flexible on applying existing graph algorithms without any change of their implementation to utilize FPGA resources.

They improve the existing models of vertex-centric approaches to have an efficient partition mechanism that relies on "interval-based" shards, where vertices are spread out across "intervals" and edges are divided among sub-shards (Figure 3). The underlying graph partitioning brings higher bandwidth due to effective utilization of data locality. Because many graph algorithms are iterative, they maintain local computation across vertices in a single interval (e.g. $I_1, I_2, ..$) per iteration (e.g. $i, i+1, ..$).

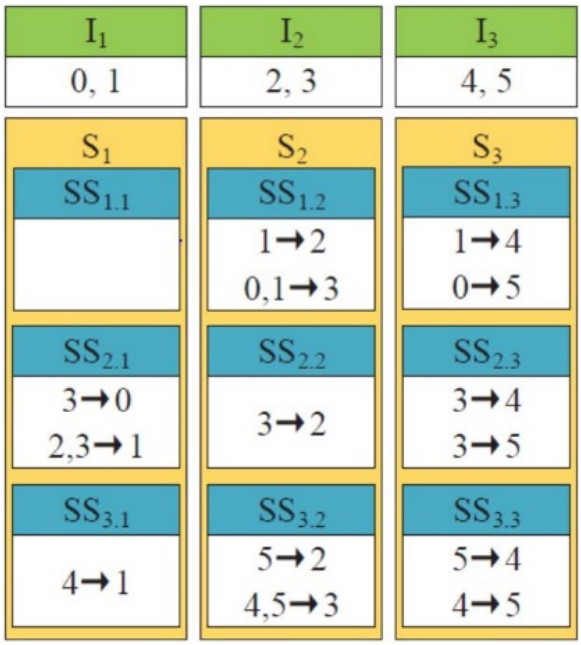

Fig. 3. Interval-shard graph partition [5]

Further, it could handle large graph sizes due to less data transfer within a single FPGA board. They utilize FPGA on-chip cache called block-RAM(BRAM), and enhance their architecture similar to SIMD processors. The system uses both local and shared memory storage to improve irregular data accesses (Figure 4). For iterative vertex-centric computation, edges data is fetched from local storages and vertices are manipulated by the controller to avoid off-chip accesses which saves a significant portion of memory bandwidth.

They evaluate FPGP memory capacity varying with graph sizes, and identify a bottleneck whenever the bandwidth of local edge storage and shared vertex memory is not identical. That concludes FPGP may not be competitive as large on-chip CPU systems, and improvements in FPGA on-chip caches are necessary to have an efficient accelerator model.

2) GraphGen [6]: GraphGen is another vertex-centric memory accelerator model which stores vertices and edges at off-chip DRAMs. Nevertheless, their goal is to provide

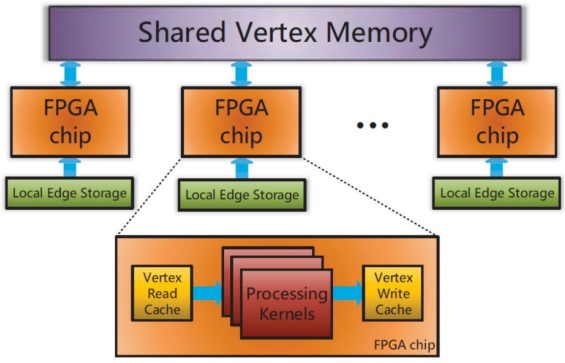

Fig. 4. FPGP Architecture [5]

a transparent design framework for developers to have a "black-box" extension over different accelerator hardwares (e.g. FPGA, GPUGPU).

They rely on an efficient mapping layer which translates vertex-centric update to a set of custom instructions (i.e. vertex program) that is capable to run on FPGA graph processors. Further, it is extended to have a SIMD version to improve the parallelism in a vertex program.

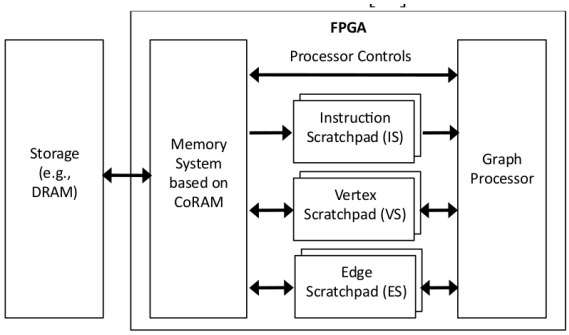

Fig. 5. GraphGen Architecture [6]

Figure 5 demonstrates the internals of GraphGen. FPGA block-RAM is used to store a subset of vertices and edges, and a local copy is kept at intermediary scratch-pads for fast retrieval. CoRAM is an interface to graph data stored at off-chip DRAM memory, and also acts as a controller to share resources for computation tasks at graph processors. As many FPGA accelerator models do, GraphGen also relies on automatic partition strategies to fit data into local scratch-pads. Then, the vertex program is executed for all vertices across the loaded data.

GraphGen has been evaluated on execution capabilities over different case-studies, but has not focused on the scalability issues. Such that off-chip DRAM is not effectively utilized over the increasing number of irregular memory accesses, and fail to maintain the minimal memory bandwidth waste.

3) GraphOps [7]: GraphOps introduces a novel hardware accelerator library for data-flow execution models targeted on FPGAs. It's beneficial not only for graph computation, but others which rely on the same hardware for different types of analytical forces. Also they present a modified data structure to enhance spatial locality and vertex level parallelism.

Here's the explanation of few building blocks of GraphOps hardware design. These blocks are considered to be common patterns found on implementing graph algorithms.

- Data handling blocks: Handle input data, reduction over the vertex neighborhood, update property set
- Control blocks: Control logic for handling the data-flow execution.
- Utility blocks: Extra logic for handling memory and host systems.

Neighborhood property reduction is key to GraphOps operation, where reduction over neighborhood property data is allowed.

Figure 6 shows the organization of GraphOps blocks to construct PageRank algorithm.

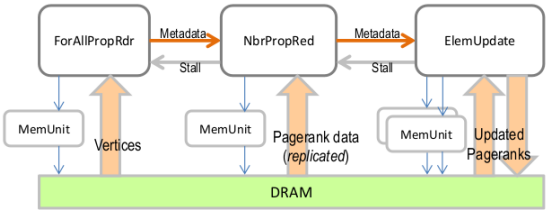

Fig. 6. GraphOps Blocks Flow [7]

GraphOps improves the execution of new score generation for vertices in PageRank algorithm, which is known to be the section of bottleneck in the literature. To update the vertex PageRank score, GraphOps enable the score reduction of neighbors' PageRank values.

GraphOps has been compared against an optimized version of C++/OpenMP code that implements PageRank algorithm. Observations include:

- Good cache locality effects at software version for small graphs, but nothing on GraphOps since FPGA doesn't reuse cache data much.
- More memory channels in both versions increase the performance
- Limitations of memory requests per neighbor over reduction of data.

GraphOps is also limited by FPGA bandwidth, but importantly perform better than it's software counterpart. Also it provides architectural building blocks to implement new graph algorithms in a data-flow execution model.

B. 3D-stacking

3D-stacking technology has been emerged as a good candidate for in-memory graph processing due to it's simplicity of putting logic and memory to a single memory unit to reduce memory bandwidth.

1) Tesseract [4]: Tesseract is a new hardware accelerator architecture that enhances 3D-stacking technology to effectively utilize available memory bandwidth and communication across memory units. Further, it specializes memory prefetching techniques to align with graph data access patterns.

Tesseract uses an alternative to 3D-stacked DRAM called Hybrid Memory Cube (HMC). HMC provides high bandwidth proportional to available memory capacity.

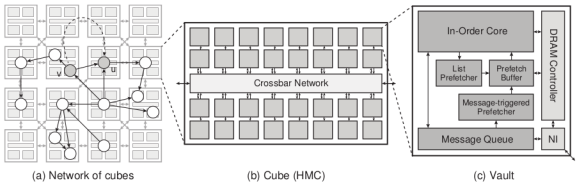

Fig. 7. Tesseract Architecture [4]

Figure 7 shows such HMC organization which has 8 DRAM layers, composed by 32 DRAM controller modules that are connected via high-speed serial links. Host CPU processors map Tesseract enabled HMC as a part of their own memory, keeping Tesseract cores to use their local DRAMs.

As Figure 7.a demonstrates, host is responsible to distribute the graph workload across HMC, enabling vertex-centric computation to be performed at each controller level. Tesseract cores use message passing to communicate with other cores. Blocking and non-blocking message passing mechanisms are exploited.

Tesseract utilizes high-memory bandwidth by adopting two memory prefetching models, and uses internal prefetch buffer to keep all prefetched data.

- List prefetching: To cater irregular and sequential vertex access patterns at graph traversals, Tesseract uses stride based prefetching technique employed with a prediction table.
- Message-triggered prefetching: To cater random access patterns at graph computation, Tesseract uses hint based message triggered technique to prefetch data from non-blocking message passing calls. Such that it argues many random accesses are performed over edge-flows, enabling remote vertices to be kept at different Tesseract cores.

Tesseract doesn't utilize any software prefetching techniques, since it relies heavily on distributed memory architecture via message passing.

In the evaluation, Tesseract validates their approach by proving it's effective usage over large internal memory bandwidth and prefetching techniques to handle irregular data access patterns. One interesting observation is the one-to-one mapping of Tesseract core with computational units called vaults. By their experimental results, it's shown that it leads to have an imbalanced load across HMC, which under-utilizes the computation power and memory bandwidth. Also Tesseract supports ideal scale-up when incresing the memory capacity, but suffer from additional message-passing overhead when the capacity is magnitude higher.

Tesseract has not focused on efficient strategies to distribute the workload across HMC in it's architecture, but has been evaluated by employing different graph partitioning algorithms. Not surprisingly, the performance is improved due to effective data locality.

C. Domain-specific

1) Graphicionado [1]: Graphicionado provides a domain-specific accelerator framework that any graph applications can be plugged into. Additionally they provide custom data-types and structures that is well suitable for vertex-centric programming that improves memory usage and parallelism.

They showcase the transparency of the system by extending the programmable pipeline of a software graph processing framework (e.g. GraphMat), and hide the internal data movement from the programmer. **Process_Edge, Reduce** and **Apply** are the basic chunks that can be used define custom computations. Figure 8 shows the phases of processing and applying graph computation on vertices.

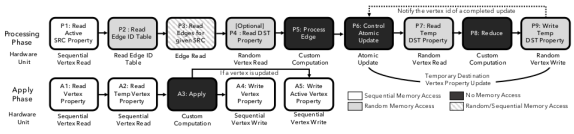

Fig. 8. Graphicionado Processing [1]

Graphicionado Pipeline Terminology:

– Vertex Read: Reading vertices are supported in both sequential and random manner.
– Edge Read: Given an edge, sequential and random read of edge data is possible.
– Process Edge: Custom computation can be defined to process on given edge.
– Atomic Update: The destination vertex is being fetched, modified and updated. Also it preserves the atomicity of the operation.

Following is the list of optimization done for basic Graphicionado pipeline presented at Figure 8.

- Improving atomic update: Graphicionado adopts a large on-chip embedded DRAM scratchpad memory to reduce the edge data access latency and to preserve atomicity.
- Adopts prefetching: To avoid many off-chip memory accesses, cache-lines are prefetched to scratchpad memory at sequential reads.
- Improvements for symmetric data-layout: Undirected networks are symmetric, such that Graphicionado improves the usage of such data-layout by avoiding extra reads when updating remote vertices.
- Dynamically sized vertex data: Large vertex properties are split into constant sized flits, and process on the fly whenever the complete data is available.
- Improve the parallelism by splitting the processing: Without replicating the pipeline to improve the parallelism, the system split the processing element into two such as source and destination vertex oriented units.

Lack of on-chip scratchpad memory does limit the size of input graph to be processed. Graphicionado employs graph slicing mechanisms. As an example, Figure 9 shows how Graphicionado slice an input graph into two slices based on the destination vertex id. Then the processing pipeline operates on a slice per iteration.

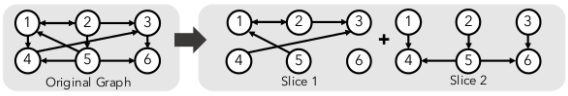

Fig. 9. Graph slicing [1]

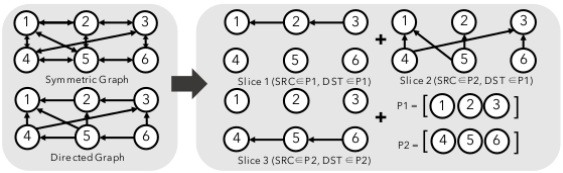

Fig. 10. Graph slicing for symmetric data layout[1]

Figure 11 visualizes a sample edge table which is used to find the edges mapped to vertex ids (Figure 11.a). To fit large edge table into the scratchpad memory, it's compressed to store a subset of edges.

Edges[] in Memory

Index	1	2	3	4	5	6	7	8	9	10	11
Edge	(1,2)	(1,4)	(2,1)	(2,3)	(2,5)	(3,6)	(4,3)	(5,1)	(5,4)	(5,6)	*

(a) EdgeIDTable

Index	1	2	3	4	5	6
EID	1	3	6	7	8	11

(b) Coarsened (2x) EdgeIDTable

Index	1	2	3
EID	1	6	8

Fig. 11. Edge Table [1]

Graphicionado has been evaluated for two classes of graph algorithms, one that accesses all vertices in it's iteration (e.g. page-rank, collaborative-filtering) and other accesses only a portion of active edges (e.g. breadth-first search, single source shortest path). We believe that brings more impact to Graphicionado evaluation, such that we could compare different data flows to be optimized for same memory system.

The hardware accelerator model has been compared with it's counterpart to software processing framework (e.g. GraphMat), and achieve a performance benefit. But the throughput of the system is dependent on the algorithm, where it actually depends on the memory access patterns. As an example breadth-first like algorithms does not gain a higher throughput as proportional to page-rank, since former depends on random portion of graph data at each iteration. Also they have identified most of the energy is spent on embedded-DRAM and it's relatively low compared with processor energy consumption. The set of optimization works well on scaling Graphicionado for large graphs with comparatively low performance degradation.

In other hand, Graphicionado tries to improve the existing memory systems available at general purpose processors without dependent on external embedded devices. Also we would like to see some experimental results of the system compared with accelerator models on top of embedded-DRAM.

2) Extended-GraphLab [8]: The authors of GraphLab, have extended their software graph processing framework to have a customizable hardware accelerator model. It's been optimized for both vertex-centric and GAS graph computation, and parallelized over an asynchronous execution model.

The proposed accelerator architecture is shown at Figure 12.

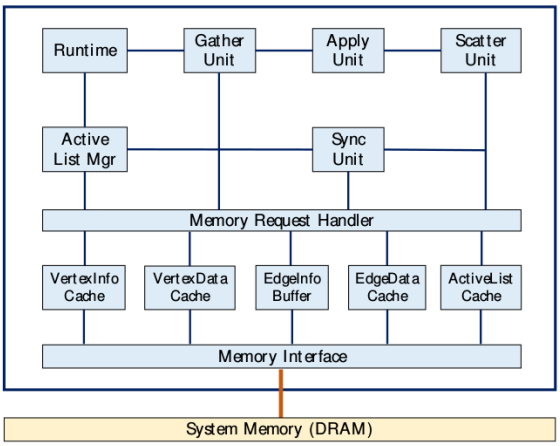

Fig. 12. Accelerator Architecture [8]

- *Runtime*: Controller on managing the number of active vertices based on the availability of system resources.
- *Gather Unit*: Fetch neighborhood data, dynamically prioritizing the allocation of vertex tasks.
- *Apply Unit*: Perform vertex operation
- *Scatter Unit*: Spread the computed results back to neighbors, and schedule future neighbors to avoid write-after-read hazards.
- *Active List Manager*: Extract vertices to process from the Active List and pass to Runtime.
- *Sync Unit*: Maintain the consistency of vertices to be processed. Figure 13 demonstrates the micro-architecture of sync unit. Sync unit preserves the sequential consistency among the vertices by assigning an unique rank to them, such that any vertex is ordered among it's adjacent nodes. Each vertex has id, rank, state and stalled requests. Using such information, neighboring vertex data (NVD) needs to ensure the ordering to avoid read-after-write hazards in the system. Content Addressable Memory (CAM) is used to identify the given vertex in the unit table.

The Compressed Sparse Row (CSR) structure has been used to store the input graph in memory, and caches could be connected to single or multiple DRAM via a memory interface.

Apart from improved performance, this study focuses on the evaluation of power, energy and estimate area for each memory block. DRAM power is significant in power consumption compared with proposed accelerator units. This is due to many

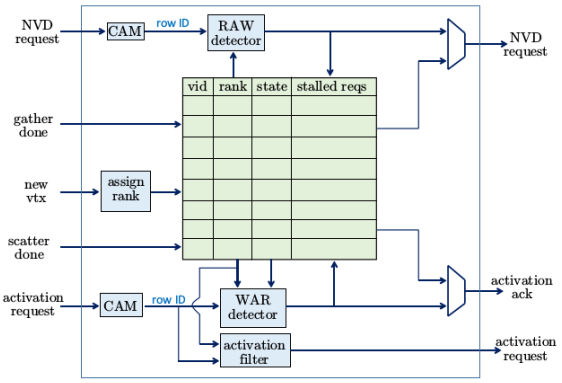

Fig. 13. Micro-architecture of sync unit [8]

data-intensive tasks happened at vertex centric computation. But overall, significant power efficiency have been achieved by the factor of 65 over the CPU models.

3) Branch-avoiding models [9]: This study brings new insights to domain specific graph accelerators by proposing strategies to avoid branches when operating on graph algorithms. Branch prediction plays a critical role over iterative graph computation on top of single or many core architecture, and a key factor for performance degrade when it's not effectively handled.

They address this problem by taking examples from well-known graph algorithms; such as Shiloach-Vishkin connected component (SV) and BFS algorithms [9]. It's experimentally shown that a significant degrade of performance incurs due to branch mis-prediction at early iterations of SV. Their optimization include a novel code transformation at assembly level for SV which reduce branch mis-prediction.

Their analysis include 2-bit predictors to deal with sequential iterative process over vertices and edges. For an example, we will briefly discuss the branch mis-prediction effect at SV algorithm (Algorithm 2).

branch prediction at SV: SV proceeds by assigning vertex ids as component ids for all vertices, and update them when iterating through adjacent neighbors such that it iterates over the conditional loop $n+1$ times in the main sequential body, where n is the number of vertices. It's shown theoretically, that finding neighbors does incur total n number of branch misses approximately, and it's heavily dependent on the distribution of input graph (e.g. scale-free). This study prevents such mis-prediction by manually intervening the variable placement from increasing the intermediary write-back states (Algorithm 3).

Their evaluation suggests a clear improvement of SV algorithm against non-optimized approach over different processors (e.g. Intel, AMD). Also they have observed that branch mis-prediction is correlated to time than memory traffic which argues the much needed architectural support to accelerate such algorithms.

Algorithm 2: Branch-based Shiloach-Vishkin algorithm for finding connect components.

```
// Algorithm initialization
for v ∈ V do
    CC_id[v] ← v
change ← 1
// Connected component labeling
while change ≠ 0 do
    change ← 0
    for v ∈ V do
        c_v ← CC_id[v]
        for u ∈ Neighbors[v] do
            c_u ← CC_id[u]
            if c_u ≤ c_v then
                CC_id[v] ← c_u
                change ← 1
```

Algorithm 3: Branch-avoiding Shiloach-Vishkin algorithm for finding connect components.

```
// Algorithm initialization
for v ∈ V do
    CC_id[v] ← v
change ← 1
// Connected component labeling
while change ≠ 0 do
    change ← 0
    for v ∈ V do
        c_v^init ← CC_id[v]
        c_v ← c_v^init
        for u ∈ Neighbors[v] do
            c_u ← CC_id[u]
            if c_u ≤ c_v then
                c_v ← c_u
        CC_id[v] ← c_v
        change ← change ∨ c_v ⊕ c_v^init
```

D. GPU

1) TOTEM [10]: TOTEM harness a hybrid version of CPU and GPU to leverage the concurrent processing of large scale partitioned graphs. Such that CPU is being utilized for fast sequential processing, while GPUs for bulk parallel processing. TOTEM try to balance the task workload between CPU and GPU to bring out the best in both worlds.

Also, this study focuses more on scale-free graphs, where we have few high-degree nodes with a lot of low-degree nodes. many low degree nodes are identical, where you can exploit SIMD parallelism over GPU multi-threading. To achieve that, TOTEM address two key challenges;

- Efficiently utilize GPU local memory and host-to-devide transfer bandwidth
- Matching SIMD architectural with graph data dependency model.

Figure 14 shows the distribution of graph data across system and device memory systems, where α and β defines the ratio of edges to be remained and crossed consequently.

It's been theoretically evaluated that the performance is influenced by the processing of slowest component of the system, which is CPU relatively. The communication overhead between two devices is negligible compared with the processing force. The speedup is dominated by GPU work, hence it

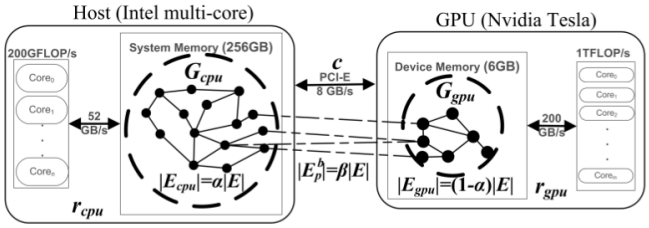

Fig. 14. TOTEM Model [10]

can be measured inverse proportionally to the portion of edges remained at CPU system.

TOTEM use Compressed Sparse Rows (CSR) structure to represent a graph (Figure 15). Vertex ids are mapped with assigned partition to fetch other neighbors ordered by local to global. CSR is known to perform at low-cost on GPUs but perform poorly for dynamic graph updates.

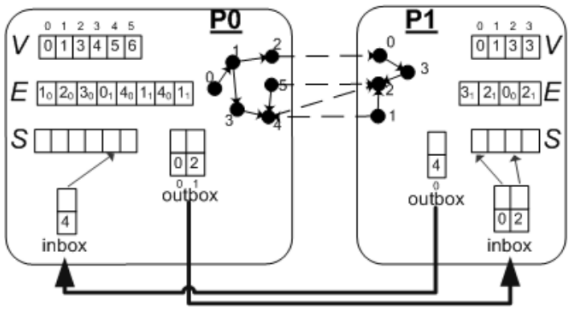

Fig. 15. Compressed Sparse Rows structure [10]

Also the system maintains two buffers per vertex that are referenced to remote neighbor and self vertex which is remote to another partition. The buffers are sorted to utilize incorporated prefetching mechanisms. Further, TOTEM map graph data with GPU memory via PCI-E bus to guarantee high-bandwidth data transfer.

Further, the processing elements of GPU and CPU are overlapped to keep balanced resource utilization. Since GPU processes tasks faster than it's counterpart CPU, the system tries to mask it by overlapping with the CPU-GPU communication overhead.

The study tries to argue the importance of coupling CPU and GPU in it's hybrid approach. It's been observed that the performance advantage acquired when GPU is capable to process any portion of the graph is comparatively low than it's proportion to the large graph. Also it's possible to design any generic graph algorithms on top of TOTEM data structure that can scale well with the increse of processing elements.

One interesting observation is related to energy cost attached to GPU processing, where the study alleviates the problem of GPU peak power consumption by it's capability of quickly convergence into an idle state. Maintaining such state saves a lot of energy in modern GPU enabled devices.

2) cuSTINGER [11]: cuSTINGER is a specialized data-structure designed on top of NVIDIA's CUDA enabled GPUs that incorporates handling temporal updates to the input graphs (STINGER as it's general-purpose counterpart [12]). It accelerates static and streaming graph computation by efficiently transferring the graph updates between memory and computational units via the proposed data structure.

STINGER keeps a subset of multiple edges in blocks where edge data is maintained in a structure. cuSTINGER specializes this array of edge structure to a block of arrays where each block is considered as a structure. This would improve data-locality in GPU where consecutive data accesses are welcomed. To avoid underutilizing the GPU processor, large edge blocks are considered to allocate for both static and dynamic graphs by cuSTINGER memory manager.

cuSTINGER supports (un)wighted adjacency lists to store graph information including any vertex or edge properties via different modes. It can switch modes at initialization to effectively manage limited GPU resources given the run time parameters.

Edge, vertex insertion and deletion are possible at cuSTINGER. The system separates former processes to get advantage over GPU parallelization and memory management. Graph updates are considered as events, and cuSTINGER supports high velocity. The granularity of such events are up the application to decide, but the study motivates to align with underlying graph algorithm behavior. Graph updates are considered to be expensive at cuSTINGER where it needs to be copied from host to device, and also vertex might need to update it's adjacency list with a new one which causes an incremental overhead. Also they identify a kernel launch overhead at initialization that causes significant performance overhead for small event batches.

While this data structure is being optimized to work with GPUs, it has been experimented in modern CPUs too, and produces exact results over triangle counting algorithm with relatively low performance overhead.

3) GunRock [13]: GunRock is a high-level graph processing library designed to harness data-centric abstraction on top of GPUs, which they introduce using a new data structure called frontier. Frontier is used as an programming interface which several graph primitives can be applied into. Three such primitives are introduced (Figure 16).

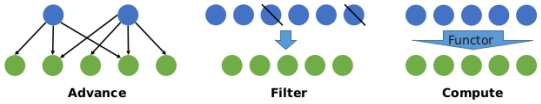

Fig. 16. GunRock data primitives [13]

- **Advance** is used to perform operations on multi-hop neighbors. Also the primitive can be parallelized for aggregation, update, fetch and new edge generation.
- **Filter** is used to find a subset of vertices or edges from the current frontier based on a programmatic criteria.
- **Compute** performs an operation over the elements at given frontier.

GPU Optimizations:

- Kernel Fusion: Integrate multiple operations asynchronously into single GPU kernel. GunRock utilizes Compressed Sparse Row (CSR) structure to treat vertex or edge data as structure of arrays.
- Workload balance: To deal with imbalanced workloads generated at Advance step, GunRock adopts Cooperative Thread Array (CTA) mechanism to interact between several GPU threads.
- Adaptive load-balancing strategy per topology to deal with dynamically sized neighborhoods.

Instead of vertex centric computations, GunRock manipulate frontiers. Bulk synchronization is supported for simplicity and performance. GunRock improves the partitioning mechanism by grouping equal number of edges together, and assigning them to blocks. But they identify that the grouping need be dynamically changed depending on the topology, where two variations including fine-grained grouping for smaller neighborhoods, and coarse-grained grouping for relative larger neighborhood have been tried out.

GunRock has been compared against both CPU and GPU based graph libraries for different classes of graph algorithms. Apart from showing significant performance results over them, GunRock has a low performance overhead when the graph is dense, such that frontiers are regularly used. Such data organization also provides relatively slow computation, whenever there are significant strides on pointer jumps.

Specially they provide flexibility to write new graph primitives from the templates written over C-like device language.

GunRock has following limitations:

- Less support for dynamic graphs.
- Neighborhood aggregation, reduction need improvements
- Kernel fusion as not as better than hardwired GPU implementations.
- Scalability issues due to GPU memory bandwidth.

4) MapGraph [14]: MapGraph is designed as a graph programming framework to harness SIMD architecture in GPUs. The goal is to cater GAS centric graph computation, which makes it more unique among other libraries on vertex centric computations.

It provides on-the-fly decisions for several optimization strategies in the run-time for scatter and gather phases as apply phase is known to be embarrassingly parallel.

- Dynamic scheduling: MapGraph assign workload to GPU threads based on a variation of the vertex degree distribution.
  1) CTA-based scattering: Each neighbor is handled by one CTA thread.
  2) Scan-based scattering: A range of neighborhood is calculated by prefix sum to form a compact scatter vector.
  3) Warp-based scattering: A variation of CTA-based scattering, but a warp is being assigned per thread to access adjacent neighbors.

  CTA-based scattering is applied for large degree distribution, followed by warp-based scattering to relatively low degree vertices. For remaining vertices, scan-based scattering is applied.
- Two-phase decomposition: Scattering is split into two phases; scheduling and computation. This scheduling groups edges by cooperating with multiple CTA threads, and then the computation phase access the same number of adjacent vertices and perform the operation.

MapGraph computation pipeline is illustrated at Figure 17. The model checks whether the frontier size has satisfied the given threshold value, such that one of optimization strategies will be applied accordingly.

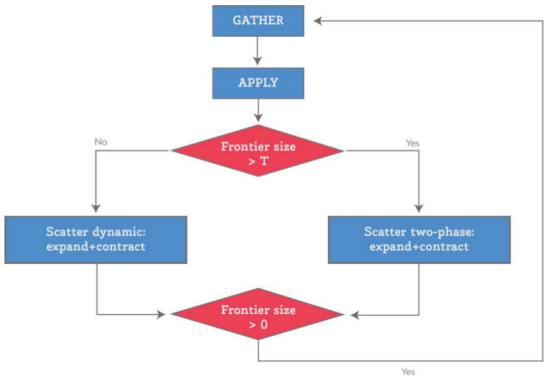

Fig. 17. MapGraph computation pipeline [14]

MapGraph introduces 3 primitives to implement any algorithm on top of the proposed computation model, such as VertexType, EdgeType and FrontierType. Also it utilizes the structure of array to represent graph data, where data-level parallelism is exploited at vertex level.

MapGraph experiments are conducted at NVIDIA Tesla K20 cores and compared against Medusa - an GPU version of popular GAS based software graph processing framework: PowerGraph. It's shown 42 times faster than naive manually optimized BFS algorithm. Also the results include a comparison with CPU-based GraphLab framework [14]. MapGraph outputs a magnitude-lower execution time for several graph algorithms, against a multi-threaded version of GraphLab.

III. Discussion

In this section, we will discuss few challenges on implementing hardware accelerator models. The discussion is motivated by a recent graph performance benchmark [15] which reveals some important observations about large-scale graph computation performed on an IvyBridge server. The computation yields comparatively low instruction throughput, due to low memory level parallelism and memory latency in modern CPUs. Further, they show that we can not hide the low memory level parallelism by increasing the number of threads. That will incur additional overheads including more cache misses, synchronization issues etc. Also it would not

save any energy since there are many stalled cores awaiting to be utilized over imbalance graph workload.

In general we discuss few pros. and cons. on implementing such accelerator models on different hardware platform, outlining key design patterns.

a) FPGA: We analyze that many FPGA systems access off-board resources via PCI-e, thus suffer from it's supported bandwidth. Large on-chip memory reduces the off-board resource usage, hence could provide more parallelism. Also scalability issues have not exploited on FPGA models, that delays large-scale deployments. One could argue former models do align with 3D-integration technology as FPGA provides customizable hardware support.

b) 3D-stacking: Integration of computation inside memory has been studied over graph domain as a new direction. Since graph includes many moving computation units, such in-memory devices need to communicate efficiently through a shared medium. Also, we argue whether cores are being effectively utilized for in-memory computation over modern day processors or GPUs, since heavy power consumption could lead to an early saturation point sooner.

Prefetching mechanisms play a critical role in in-memory 3D-stacking technology. We believe exact prefetching is hard to achieve, due to it's dependency with the given graph algorithm. As an example, all graph centrality algorithms might not follow data-locality, where such as Eigenvector centrality depends on the spectral properties of the whole graph.

c) GPU: Large graphs need more space to store. Undoubtedly single GPU can not cater them all where they need to be scaled up to a cluster of GPUs. But it incurs additional complexity as communication between multiple GPUs is identified to be difficult. Also the distribution and partition of graph data across GPUs is non-trivial as data usually have multiple dependencies among vertices. Several studies suggest the key to success on GPU graph processing is to overlap communication and computation as much as possible [16]. Effective utilization of GPU resources is a hard problem to tackle, neither under or over utilization should not be permitted.

Due to the inherent complexity of modeling general graph computation over GPUs, the domain has been incrementally studied over several classes of algorithms [17], [18], [19]. Also, handling atomic updates over graph structures is not properly studied in SIMD architecture. and the synchronization patterns over multiple GPU cores would bring more traffic towards host communications (e.g. kernel invocations).

d) Optimized data structures: Many studies exploit different data-structures to be optimized for hardware accelerators. While compressed sparse row (CSR) is being more popular among literature, some exploit coordinate lists and ELLPACK too [7].

As Figure 15 shows, CSR includes a vertex, edge and property arrays. This data organization does not bring benefit always. As an example, PageRank algorithm require to fetch neighbor property data simultaneously, which may overflow from the scattered property requests. Such that locality needs to be improved to cater specific algorithms. Also it's important to optimize the given data structure to support underlying graph abstraction too.

e) Load balancing strategy: Apart from two graph abstractions(i.e. vertex-centric, gather-apply-scatter) we have discussed, there do exist alternative models exploited in the literature. Such abstractions clearly dominate the efficiency of load balance strategies. Message-passing is such abstraction model, where edges and vertices send messages to adjacent nodes in order to distribute the computation. This approach causes heavy workload imbalance on GPUs for many real world graphs [13].

CPU strategies to handle the burst workload rely more on task-parallelism, where execution of tasks performs parallel but also in speculative manner. Ligra and Galois are such models in the literature [20]. Load balancing in GPU brings severe challenges too, including synchronization issues, locking overheads and dynamic data structure support.

f) Evaluation strategy: Many proposed accelerator models are not generic, in the sense they are optimized for specific classes of graph algorithms. Following algorithms including Single source shortest path (SSSP), Page-rank, Breadth first search (BFS) ,Collaborative filtering (CF), Stochastic gradient descent (SGD), Loopy belief propagation, Vertex Cover (VC), Clustering coefficient (CC) etc. are usually tried out for benchmark.

Since accelerator models try to be effective for irregular access patterns on graphs, they consider a variety of such patterns to demonstrate system behavior. Such that, algorithms like PageRank and collaborative filtering do need all vertices to be participated in a single iteration while others do require only a subset. Also, we believe such classes of iterative nature do cover the breadth of given approach. Overall, they haven been evaluated against many software graph processing frameworks (e.g GraphMat, GraphLab, Medusa), and achieve a significant performance gain, specially over the effective utilization of memory bandwidth.

In summary, Table I details a summary of comparison of hardware accelerator models studied in this work.

IV. Conclusion

Graph based accelerator models have been risen to overcome challenges found in general data-flow execution models. Recent studies suggested on designing such acceleration, one could require the support of underlying hardware models. In this study, we outline few accelerator models over FPGA, 3D-stacking and GPU. Also we explain some algorithmic optimizations that could be exploited along with such models. Our discussion ranges to two key graph abstractions (e.g. vertex centric and GAS) over different classes of graph problems. Further, we highlight useful design patterns and strategies found on recent studies. Experimental results have been discussed with the special consideration to energy consumption.

We believe the recent attraction over hardware accelerator models on large scale graph processing will open new research

Model	Platform	Graph Abs.	Data-structure	Memory	Partitioning	Load balancing	Evaluated Algos.	SW interface
FPGP [5]	FPGA	Vertex-centric	Sharded List	Block-RAM, DRAM	Interval-based	No	BFS	No
GraphGen [6]	FPGA	Vertex-centric	List	Block-RAM, DRAM	Tile-mapping	No	Applications	Yes
GraphOps [7]	FPGA	Data-flow	Locality-optimized array	DRAM	No	Yes, parametric	PageRank BFS SSSP	Yes
Tesseract [4]	3D-stacked	Vertex-centric	List	3D-stacked DRAM	Yes Edge partioning	Yes	Conductance,Vertex-cover	Yes
Graphicionado [1]	CPU	Vertex-centric	Coarsened Edge Table	DRAM	Yes Graph slicing	Yes	BFS,PageRank,SSSP	Yes
Extended GraphLab [8]	CPU	Vertex-centric, GAS	Compressed Sparse Row	DRAM	Yes	Yes	BFS,PageRank,SSSP	Yes
TOTEM [10]	CPU, GPU	Vertex-centric	Compressed Sparse Row	GPU-mem DRAM	Yes	Yes,overlapped CPU, GPU exec.	BFS PageRank SSSP	Yes
cuSTINGER [11]	GPU	Vertex-centric	cuSTINGER array	GPU-mem	No	No	Triangle-counting	Yes
GunRock [13]	GPU	Vertex-centric	Frontier	GPU-mem	No	Yes	BFS,PageRank,CC	Yes
MapGraph [14]	GPU	GAS	Frontier	GPU-mem	No	Yes CTA	BFS,SSSP,CC	Yes

TABLE I
A SUMMARY OF HARDWARE ACCELERATOR MODELS

directions. Along with graph models, we will extend our study to cover the data-flow execution models in the future. Such that many software frameworks could be utilized over their rich data semantics in hardware designs. Moreover, the integration of different platforms could be extended to have advanced hardware designs, and such exploration would enable more design space for algorithm developers and programmers.

Also, such models could be influential on developing energy saving chips. Specializing hardware design over application specific knowledge would bring more opportunities to overcome the power limitation.

ACKNOWLEDGMENT

We would like to thank Dr. Oded Green, Georgia Institute of Technology for his diligent guidance to narrow down this study, and also the authors who share their papers currently under journal review.

REFERENCES

[1] T. Jun, H. Lisa, W. Narayanan, S. Nadathur, and S. Margaret, "Graphicionado : A High-Performance and Energy-Efficient Accelerator for Graph Analytics," *49th International Symposium on Microarchitecture*, vol. To Appear, Oct. 2016.

[2] 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*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[3] 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, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: http://dx.doi.org/10.14778/2212351.2212354

[4] J. Ahn, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 105–117, 2015.

[5] G. Dai, Y. Chi, Y. Wang, and H. Yang, "Fpgp: Graph processing framework on fpga a case study of breadth-first search," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 105–110. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847339

[6] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," in *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 25–28. [Online]. Available: http://dx.doi.org/10.1109/.13

[7] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 111–117. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847337

[8] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 166–177, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/3007787.3001155

[9] O. Green, M. Dukhan, and R. Vuduc, "Branch-avoiding graph algorithms," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '15. New York, NY, USA: ACM, 2015, pp. 212–223. [Online]. Available: http://doi.acm.org/10.1145/2755573.2755580

[10] A. Gharaibeh, E. Santos-Neto, L. B. Costa, and M. Ripeanu, "Efficient large-scale graph processing on hybrid CPU and GPU systems," *CoRR*, vol. abs/1312.3018, 2013. [Online]. Available: http://arxiv.org/abs/1312.3018

[11] O. Green and D. Bader, "custinger: Supporting dynamic graph algorithms for gpus," in *IEEE High Performance Extreme Computing Conference*, 2016.

[12] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE, 2012, pp. 1–5.

[13] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," *CoRR*, vol. abs/1501.05387, 2015. [Online]. Available: http://arxiv.org/abs/1501.05387

[14] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, ser. GRADES'14. New York, NY, USA: ACM, 2014, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/2621934.2621936

[15] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, ser. IISWC '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 56–65. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2015.12

[16] J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1068–1079. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.102

[17] J. Soman and A. Narang, "Fast community detection algorithm with gpus and multicore architectures," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 568–579. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.61

[18] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek, "Betweenness centrality on gpus and heterogeneous architectures," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6. New York, NY, USA: ACM, 2013, pp. 76–85. [Online]. Available: http://doi.acm.org/10.1145/2458523.2458531

[19] A. Polak, "Counting triangles in large graphs on GPU," *CoRR*, vol. abs/1503.00576, 2015. [Online]. Available: http://arxiv.org/abs/1503.00576

[20] S. Tzeng, B. Lloyd, and J. D. Owens, "A gpu task-parallel model with dependency resolution," *Computer*, vol. 45, no. 8, pp. 34–41, August 2012.