2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Energy Efficient Architecture for Graph Analytics Accelerators

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

***Abstract*—Specialized hardware accelerators can significantly improve the performance and power efficiency of compute systems. In this paper, we focus on hardware accelerators for graph analytics applications and propose a configurable architecture template that is specifically optimized for iterative vertex-centric graph applications with irregular access patterns and asymmetric convergence. The proposed architecture addresses the limitations of the existing multi-core CPU and GPU architectures for these types of applications. The SystemC-based template we provide can be customized easily for different vertex-centric applications by inserting application-level data structures and functions. After that, a cycle-accurate simulator and RTL can be generated to model the target hardware accelerators. In our experiments, we study several graph-parallel applications, and show that the hardware accelerators generated by our template can outperform a 24 core high end server CPU system by up to 3x in terms of performance. We also estimate the area requirement and power consumption of these hardware accelerators through physical-aware logic synthesis, and show up to 65x better power consumption with significantly smaller area.**

I. Introduction

With the end of Dennard scaling, computing systems are becoming increasingly power limited. New transistor technologies allow packing more logic in a chip, but only a small fraction of available logic can be used at a given time due to power limitations, a phenomenon known as dark silicon. This allows adding custom hardware accelerators targeted for specific tasks and that are significantly more efficient in terms of power and performance. It has been shown that accelerator rich architectures can lead to significant improvements through customizations for specific tasks.

Another motivation for customization is the increasing prevalence of cloud computing and large server farms that execute a small set of workloads repeatedly. Significant power and performance gains can be achieved by customizing these servers for the frequently executed workloads.

Many existing works focus on accelerating compute-intensive tasks using programmable hardware (e.g. GPUs, CPU vector extensions such as SSE and AVX) or custom hardware. A common characteristic of these applications is the regularity and the abundance of data and thread level parallelism. In this paper, we focus on a certain class of graph analytics applications with *irregular* execution patterns that make them hard to accelerate using existing platforms.

Specifically, we focus on iterative graph-parallel applications with asynchronous execution and asymmetric convergence. It has been shown in [1] that many graph analytics applications have such execution patterns, and can be represented with a vertex-centric abstraction model. Using this abstraction model, the authors have proposed a software framework called GraphLab to make it easy for domain experts to develop parallel and distributed programs. While the domain experts provide the application-level data structures and serial operations per vertex, the underlying software framework handles system-related complexities including scheduling, communication, synchronization, and reliability.

Our objective in this paper is similar, but targeted for architecture and hardware development of graph analytics accelerators. We propose a customizable architecture template that is specifically optimized for the target class of graph applications. We implement the common operations (such as memory access, communication, synchronization, etc.) in the proposed template. The architects/designers can plug in application-level data structures and operations into this template and generate the hardware implementation easily. This enables exploration and implementation of hardware accelerators for a large class of graph applications.

Our main contributions in this paper can be summarized as follows:

- We propose an architecture specifically optimized for vertex-centric, iterative, graph-parallel applications with irregular access patterns and asymmetric convergence. Our architecture supports asynchronous execution, which is known to be more work-efficient than bulk-synchronous execution [1, 2, 3].
- We provide cycle-accurate and synthesizable SystemC models that implement the proposed architecture template. It is possible to plug in application-level data structures and operations to easily generate hardware accelerators for different graph applications.
- We provide an experimental study that compares the area, power, and performance of the generated hardware accelerators with CPU implementations. Our area and power values are obtained through physical-aware RTL synthesis of the functional blocks using industrial 22nm libraries.

The rest of the paper is organized as follows. Section II discusses the limitations of the existing CPU and GPU architectures for irregular graph applications. The abstraction model and the proposed architecture are described in Sections III and IV, respectively. Our experimental setup is outlined in Section V, and our experimental results are reported in Section VI.

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.24

II. Irregular Graph Applications

Common characteristics of iterative graph-parallel applications have been studied recently [1, 2], and the desired architectural features have been identified [3] as follows.

Asymmetric Convergence: The number of iterations each vertex needs to be processed before convergence may vary significantly for graph analytics applications [1]. For example, it was observed that the *PageRank* algorithm converges in 77 iterations when executed on the *soc-LiveJournal* benchmark, where 51% of the vertices converge in 36 iterations, and 99.7% converge in 50 iterations [3]. It was also shown that maintaining the set of active vertices for PageRank improves work efficiency by almost twice compared to processing all vertices in every iteration.

Asynchronous Execution: A bulk-synchronous iterative algorithm has well-defined iterations that are separated by barriers. In the context of graph algorithms, when a vertex needs to access the data of its neighbors, it accesses the data computed in the previous iteration. In contrast, there are no well defined iterations in an asynchronous algorithm, and the vertices can access the latest data computed by their neighbors. It was shown that asynchronous execution converges much faster than synchronous execution for many graph applications [1]. For example, the PageRank application is shown to converge twice faster in the asynchronous mode [4].

Although more work efficient, an asynchronous implementation may run slower because of potential synchronization overheads. Race conditions are possible because the data updated by a vertex may be read simultaneously by its neighbors. This is in contrast to synchronous execution, where the readers and writers are guaranteed to be separated by barriers. Furthermore, a more strict sequential consistency property[1] may be needed for some algorithms to achieve faster convergence (e.g. Alternating Least Squares) or to guarantee correctness (e.g. Gibbs Sampling) [1].

Memory Access Bottlenecks: It is known that memory access can be the main bottleneck for graph analytics workloads [5]. The main reason is that a small amount of computation is typically performed per vertex and edge, but a large number of vertices and edges need to be processed for large graphs. A vertex/edge processed is unlikely to be processed again before most of the other vertices/edges are processed, which leads to poor temporal locality. Furthermore, for real-life unstructured graphs, the data of neighboring vertices are unlikely to be in the same cache lines, leading to poor spatial locality. As a result, each access to vertex or edge data can incur long latency to the system memory.

Load Imbalance: Vertex degrees of real graphs (e.g. social networks) follow the Power law distribution [2, 6, 7], where a small percent of vertices cover most of the edges. Assigning vertices to threads statically can lead to severe load imbalances due to the *scale-free(Power Law degree distribution)* nature of the real graphs.

A. Limitations of General Purpose CPUs

A recent performance study has shown that even the best serial and parallel implementations of graph algorithms execute instructions on an IvyBridge server at surprisingly low IPCs (most below 1.0 and many below 0.5) [8]. The authors concluded for graph applications that 1) memory latency is the main performance bottleneck, 2) low memory level parallelism (MLP) leads to under-utilization of the DRAM bandwidth, and 3) overall performance generally scales linearly with memory bandwidth consumption because of overlapped access latencies.

For a single OOO core, the maximum number of outstanding memory requests is bounded by the number of miss-status holding registers (MSHRs), which is equal to 10 for an IvyBridge core. On the other hand, for a DRAM with 90ns latency, 64GB/s bandwidth, and 64B access granularity, we need to have at least 90 outstanding memory requests to fully utilize the available DRAM bandwidth. In contrast, the authors of [8] have shown that most graph processing workloads sustain far less than 10 outstanding memory requests per core due to instruction window size limitations. Furthermore, they showed that simply increasing the number of hardware threads per core through simultaneous multi-threading (SMT) is not sufficient to improve MLP (and hence performance) substantially. More threads necessitate more hardware resources (e.g. registers) and increase cache misses, synchronization overheads, and load imbalance penalties.

Using multiple cores can allow better bandwidth utilization but reduces the energy efficiency of computation by increasing the number of stalled cores. The main problem here is that general purpose CPU architectures rely on caches to hide long memory latencies, assuming that most workloads have reasonable data locality. However, this is not the case for graph analytics applications.

Furthermore, multi-core CPU architectures incur synchronization overheads in the asynchronous mode of execution. For example, it has been shown that the Graphlab implementation of PageRank slows down by more than an order of magnitude on a multi-core system when sequential consistency property is enabled [3]. Even without sequential consistency, the computation throughput (measured as the number of edges processed per second) of asynchronous execution has been shown to be about 50% lower than synchronous execution in the same study. As a result, the work efficiency advantages of asynchronous execution do not always translate to lower execution times on today's multi-core systems.

B. Limitations of Throughput Architectures

Throughput-oriented architectures have three key features: simple cores, extensive multi-threading, and single-instruction multiple-data (SIMD) execution [9]. Mainstream GPUs today

[1] A parallel execution is defined to be sequentially consistent if and only if it is guaranteed to be equivalent to an execution where vertices are processed in some sequential order.

are such throughput architectures. They consist of multiple streaming processors (SMs), each of which is capable of executing many threads. Threads are organized into warps (or wavefronts), where execution within a warp happens in a SIMD fashion. GPUs hide long memory access latencies by scheduling thousands of threads. However, iterative graph algorithms require synchronization and atomic access to common data structures. Efficient global synchronization among thousands of threads is not supported in today's GPUs, and may require separate kernel invocations (hence frequent communications with the host). It has been shown that the amount of interaction between GPU and CPU is an order of magnitude larger for irregular graph applications compared to regular applications [10].

Due to asymmetric convergence, the set of active vertices can change significantly during execution. Statically assigning vertices to GPU threads leads to under-utilization of compute resources. Furthermore, it is hard to implement an efficient data structure that keeps track of the active vertices, because many threads need to write to it after every vertex update.

Asynchronous execution requires fine-grain synchronization between neighboring vertices, which is not well-suited for GPU architectures due to expensive locking mechanisms among thousands of threads. Typical GPU implementations of graph algorithms use the synchronous model and cannot take advantage of the work efficiency of asynchronous execution.

The SIMD nature of GPU execution leads to both control and memory divergence due to irregularity of graph applications. For example, a scale-free graph can have some vertices connected to thousands or millions of edges, while other vertices are connected to only tens of edges. Assigning vertices to GPU threads can lead to severe load imbalances within warps. For example, a recent performance study has shown that the warp utilization ratio is less than 25% for irregular graph applications [10].

Similarly, the memory access patterns are irregular for unstructured graphs. In contrast, GPUs achieve their peak memory bandwidth only when accesses are coalesced. Frequent random memory accesses in graph applications lead to under-utilization of faster shared memories [10], uncoalesced global memory accesses, bank conflicts [11], DRAM latency divergence within warps [12], and hence under-utilization of the available memory bandwidth. A simulation-based performance study on irregular GPU kernels has shown that only less than 16% of the GPU cycles are fully utilized in the graph applications studied, and the biggest performance bottlenecks are load/store unit pipeline stalls and memory access latencies [13].

III. Graph-Parallel Abstraction

There have been several graph frameworks proposed in the last few years to make it easy to develop parallel and distributed software for graph-parallel applications. Some examples are Pregel [14], GraphLab [1], Giraph [15], CombBLAS [16], SociLite [17], and Galois [18]. The readers can refer to [19] for an extensive comparison of these frameworks. In this paper, we focus on the vertex-centric ("think like a vertex") abstraction model that consists of Gather-Apply-Scatter (GAS) functions as in GraphLab. In this model, the users need to define basic data structures corresponding to each vertex/edge and implement serial functions for the following operations:

- Gather: Collect and accumulate data from the neighboring vertices and edges.
- Apply: Perform the main computation for the input vertex using the Gather results.
- Scatter: Distribute the vertex data computed in Apply to neighbors. Determine whether to schedule the neighboring vertices for future execution.

The GraphLab software framework enables asynchronous execution with sequential consistency. It also keeps track of the set of active vertices to avoid processing converged vertices unnecessarily. We have chosen the GraphLab abstraction model because many graph analytics applications can be naturally represented by this model, and there have been ongoing efforts to map emerging workloads to it.

```
PageRank(Input graph: (V, E))
1.  for each unconverged vertex $v \in V$ do:
2.      $sum = 0$                                        // gather_init()
3.      for each vertex $u$ for which $(u \to v) \in E$
4.          $sum = sum + \frac{r_u}{d_u}$                // gather_edge()
5.      $r_v^{new} = \frac{(1-\alpha)}{|V|} + \alpha \cdot sum$   // apply()
6.      doScatter = $|r_v^{new} - r_v| > \varepsilon$    // apply()
7.      $r_v = r_v^{new}$                                // apply()
8.      if doScatter then
9.          for each vertex $w$ for which $(v \to w) \in E$
10.             activate $w$                             // scatter_edge()
```

Figure 1: Pseudo-code of the PageRank algorithm

The exact programming interface for our architecture template is not included in this paper due to page limitations. However, it will be published in the future in another publication. The application specific data structures and functions in the programming interface are clearly separated from the architecture template implementation. The PageRank algorithm is given in Figure 1 as an example. To implement PageRank on our template, a user can define the data structure associated with a vertex as a pair of fixed-point values, corresponding to 1) one over vertex degree ($1/d_v$) and 2) vertex rank (r_v). Then, the user needs to fill in the pre-defined functions corresponding to different GAS operations, as shown in the comments of Figure 1. For example, *gather_edge* function consists of a simple multiply-add operation (line 4), while the *scatter_edge* function sets a predicate parameter based on the result of *apply* function to indicate if the neighboring vertex needs to be activated (line 10).

All application-specific data structures and functions are defined in plain C language, and are plugged into our architecture template. The template automatically removes the hardware corresponding to empty data structures and unused features. As an example, the application-specific part of our PageRank implementation is about 20 lines of C code, while the common architecture template is more than 25,000 lines of SystemC code, and not visible to the user.

If the objective is architecture simulation, the users can specify the latency and throughput of each GAS function. RTL can also be generated for each GAS function through High-Level Synthesis (HLS). All other operations involving memory access, synchronization, communication, etc. are implemented in the provided architecture templates, and they are parameterizable based on application requirements.

IV. Proposed Architecture

We propose a templatized architecture that can perform the operations defined in Section III and that is specifically optimized for graph-parallel applications that have the execution patterns outlined in Section II. Its main features can be summarized as follows:

1) Tens of vertices and hundreds of edges are processed simultaneously to achieve high levels of memory-level parallelism. This is done by maintaining partial states for multiple vertices and edges while waiting for responses to long-latency memory requests (Sections IV-A and IV-C).
2) Scale-free graphs are handled through dynamic load balancing. For example, hundreds of edge states can be assigned to a single high-degree vertex or can be distributed to multiple low-degree vertices during execution (Sections IV-A and IV-C).
3) Synchronization between concurrently processed vertices and edges is done in the Sync Unit (SYU) module, which is specifically designed for graph processing (Section IV-D). This module ensures sequential consistency with negligible performance overhead. Furthermore, it works in a distributed fashion without a centralized bottleneck (Section IV-H).
4) The set of active (not-yet-converged) vertices is maintained by the Active List Manager (ALM) module (Section IV-E). This module enables simultaneous high-throughput reads and writes from/to the distributed Active List (AL) data structure without the need for expensive locking mechanisms.
5) The memory subsystem is optimized for sparse graph data structures (Section IV-G).

The proposed accelerator is loosely-coupled with the host processor and it is connected to the system DRAM. It is assumed that the host processor will populate the graph data in DRAM, and send a start signal to the accelerator. Once the accelerator finishes computation, it will send a signal back to the host.

Figure 2 illustrates the proposed high-level architecture for a single accelerator unit (AU). For simplicity of presentation, we will first focus on the execution of a single AU, and then describe how to combine multiple AUs to achieve higher performance.

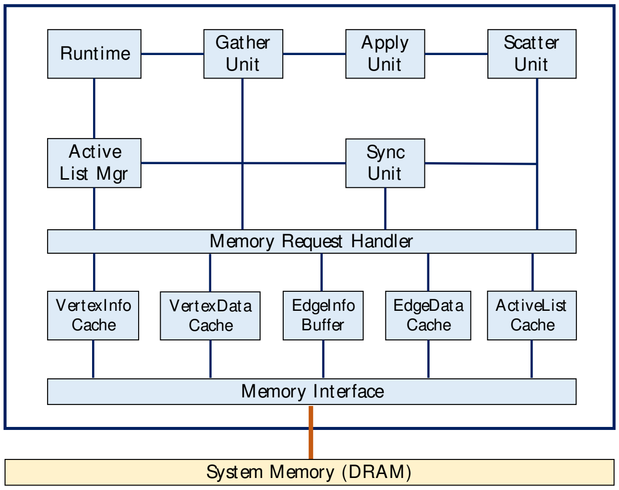

Figure 2: Single Accelerator Unit (AU) connected to the system DRAM. For clarity of the figure, not all connections between the blocks are shown.

The Active List (AL) contains the set of vertices that need to be processed. The execution begins with Active List Manager (ALM) extracting vertices from AL, and sending them to Runtime (RT). RT controls how many vertices can be processed at a given time based on the resource availability in the rest of the system. If there are enough resources, RT starts the execution of a vertex by sending it to the Sync Unit (SYU), which is responsible for the sequential consistency between the vertices that are being executed concurrently in the system. SYU assigns a rank to each vertex, and sends it to the Gather Unit (GU). GU loads the data associated with each vertex. Then it iterates over all incoming edges of a vertex, and accumulates the data specified by the application. After the Gather operations are finished for a vertex, the data associated with it is sent to the Apply Unit (APU), where the main computation for the vertex is typically performed. After that, the data computed in APU is sent to the Scatter Unit (SCU) to be distributed to the neighbors based on application specifications. SCU is also responsible for scheduling neighbors for future execution if necessary. At any given time, there are typically tens of vertices and hundreds of edges being processed by GU and SCU. All data accesses with potential race conditions must go through SYU to ensure that all vertices and edges are processed in a sequentially consistent way. Furthermore, SYU and ALM coordinate together to make sure that there are no duplications and unnecessary additions to AL.

The details of each block are described in the following subsections.

A. Gather Unit

The Gather Unit (GU) implements the Gather Program for each vertex *v*. Collecting and accumulating data from neighbors requires several memory load operations, each of which can have long latency to the system memory. For this reason, we propose a latency tolerant architecture for the GU, where many vertices and edges are processed concurrently, and partial states are stored locally. Let gv denote the number of vertices and ge denote the number of edges that can be processed concurrently in GU. In this case, GU needs to store gv partial vertex states and ge partial edge states. These values need to be set based on the application data structures, access locality, and the latency to the system memory.

The limited local storage available in GU is shared among all concurrently processing vertices. In the GU microarchitecture we propose, a credit based mechanism is used to assign the available edge slots dynamically to multiple vertices. The vertices that are supposed to execute logically before others (see Section IV-D) are given higher priority during this assignment. For example, it is possible for a high-priority and high-degree vertex to be assigned all available edge slots. It is also possible for multiple low-degree vertices to share the available storage. These decisions are done dynamically based on vertex degrees and vertex priorities.

B. Apply Unit

The Apply Unit (APU) is the module that performs computation for each vertex using the data obtained in the Gather State. The user defined Apply function operates on the data received from Gather Unit. There is no access to the system memory. The computation in this stage is typically pipelined over multiple cycles so that different vertices can be processed at different pipeline stages.

C. Scatter Unit

Scatter Unit (SCU) implements the Scatter Program for each vertex v. The application specific Scatter functions determine how to distribute the updated data of v to its neighbors. Similar to GU, multiple vertices and edges are processed in parallel to hide memory access latencies, and a credit-based mechanism dynamically assigns local storage to vertices.

For each out-neighbor u of vertex v, the application-specific function also determines whether v should activate u (i.e. schedule u for future execution) or not. An *activate* message is sent from SCU to SYU for each out-neighbor of v with two potential purposes: 1) to schedule the neighbor for future execution, and 2) to prevent WAR hazards. Even if the neighboring vertex u is not supposed to be activated, an activate message with *false* flag is sent to SYU for the purpose of preventing hazards. SCU does not write the vertex data of v or the data of the edge between u and v until an acknowledgement message is received from SYU. If SYU detects a WAR hazard, the acknowledgement message is not sent back to SCU until the WAR hazard is resolved.

D. Sync Unit

The Sync Unit (SYU) is the critical module that allows race-free and sequentially consistent execution of all vertices in the proposed architecture. SYU is in charge of coordination between vertices such that read-after-write (RAW) and write-after-read (WAR) dependencies are respected and no redundant activation occurs. The high level microarchitecture of SYU is illustrated in Figure 3.

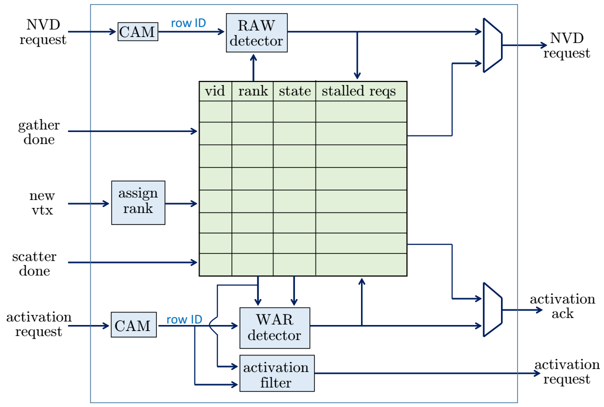

Figure 3: High level microarchitecture of SYU

The basic idea to ensure sequential consistency is to assign a unique *rank* value to each vertex before it begins execution. The rank values are increased monotonically so that the vertices that start execution earlier have lower ranks and higher priorities. We use the *edge consistency* model [1], which implements sequential consistency by enforcing ordering between adjacent vertices, because a vertex is allowed to update only its own data and the data of edges connected to it. We briefly describe the basic operations in the SYU microarchitecture below.

Maintain vertex states: Once a new vertex is received from Runtime, it is assigned a unique rank, and stored in a table, which contains all vertices currently being executed in the AU. The row corresponding to vertex v contains its ID, rank, execution state, and all stalled requests for v (see the paragraphs below). The execution state of v is also updated when *gather-done* or *scatter-done* message is received.

Maintain RAW ordering: Consider an edge $e : u \rightarrow v$ where $rank(u) < rank(v)$, i.e. the execution of u should (logically) happen before v. Sequential consistency dictates that v should not read the data of vertex u or edge e before u updates them. As shown in Figure 3, the neighboring vertex data (NVD) requests from Gather Unit (GU) go through SYU to ensure this ordering.

Assume that GU sends an NVD request for vertex u while processing edge $u \rightarrow v$. Once SYU receives this request, a small content addressable memory (CAM) is used to check if u is in the SYU table. If so, its rank is compared with the rank of vertex v. A RAW dependency is detected iff $rank(u) < rank(v)$. In that case, index v is stored in the row corresponding to u, and the request is stalled until u finishes execution[2]. If u is not found in CAM or if $rank(u) > rank(v)$, then the NVQ request is sent out to the memory interface.

Maintain WAR ordering: Consider edge $u \rightarrow v$. There is a potential WAR dependency between u and v iff $rank(u) > rank(v)$. To maintain WAR ordering, the Scatter Unit (SCU) sends an *activation message* to SYU corresponding to each

[2] There can be multiple stalled requests for vertex u from different vertices. The row corresponding to u stores all such requests. When u finishes its execution, the stalled requests are released in consecutive cycles.

edge $e : u \rightarrow v$, and waits for acknowledgement before it writes data associated with e or u. The WAR detection and prevention mechanism is similar to the RAW-related mechanism described above.

Avoid Unnecessary Activations: An activation message received from SCU contains a flag indicating whether the target vertex should be activated or not. Consider an activation message corresponding to edge $u \rightarrow v$ with a true flag. This implies vertex v should be added to the Active List (AL) for future execution. However, if vertices u and v are being executed concurrently, activation of v may be unnecessary depending on the vertex ranks. Specifically, if $rank(u) < rank(v)$, sequential consistency mechanisms guarantee that vertex v will access the data most recently updated by vertex u. So, it is unnecessary to schedule v for future execution again. SYU filters out such unnecessary activations before passing the activation requests to the Active List Manager.

Each task above is implemented as a separate pipeline in SYU. Despite the interdependencies between different pipelines, our implementation ensures high-throughput processing of requests so that SYU does not become the performance bottleneck. Note that the CAM structure in SYU is guaranteed not to overflow because the total number of vertices concurrently executed in the system is limited and controlled by the Runtime module. The low-level implementation details are omitted due to page limitations.

E. Active List Manager

The Active List (AL) stores the set of vertices that need to be executed in the future. The initial AL is application-dependent and is part of the input data. Since the AL can potentially contain all vertices in the input graph, it needs to be stored in the system memory. As explained before, the application-specific convergence condition is checked in SCU to determine which vertices to schedule for future execution, while the unnecessary activations are filtered out in SYU. The Active List Manager (ALM) is responsible for the following tasks: 1) Extract vertices from AL, and send them to Runtime for execution. 2) Receive new activation requests from SYU, and add them to AL while avoiding duplications.

For storage and data access efficiency, the AL consists of two data structures: 1) A bit vector where each bit corresponds to the presence or absence of a vertex in AL. 2) A queue of bit vector indices where each index corresponds to a 256-bit segment of the bit vector.

For the purpose of extracting new vertices for execution, ALM reads the next bit vector index from the AL queue, and loads the corresponding 256-bit segment of the bit vector. Then, it starts sending the vertices that has set bits in the bit vector to Runtime for execution.

When ALM receives an activation request for vertex v, it first checks whether the bit corresponding to v is locally stored in ALM. If so, it simply sets that bit locally. Otherwise, it sends the request to the AL memory unit. Special care needs to be taken to handle in-flight bit vectors and vertex indices. Specifically, when a vertex index is sent to Runtime, it also needs to be registered with Sync Unit, and an acknowledgment needs to be received before removing the corresponding bit from the local storage of ALM. Otherwise, an incoming activation request for the same vertex may fail to detect that the vertex is already being executed. Similarly, the in-flight bit vectors between ALM and AL memory need to be handled with care to avoid adding duplicate vertices to AL.

Similar to SYU, there are multiple pipelines being processed concurrently. The microarchitecture of ALM ensures that the dependencies between these pipelines do not cause race conditions. Furthermore, each pipeline can operate at full throughput regardless of the inter-pipeline dependencies.

F. Runtime

The Runtime (RT) module is in charge of monitoring available resources in AU and scheduling new vertex executions. It reads new vertices from ALM, and sends them to SYU when it detects that there are available resources. It is also responsible for detecting termination condition and sending out completion signal when there are no in-flight or executing vertices and AL is empty. RT is a simple module consisting of two counters to keep track of the number of vertices in Gather and Scatter stages.

G. Memory Subsystem

There are different data structures that need to be accessed when a vertex program is executed. In this paper, we assume that the popular Compressed Sparse Row (CSR) format is used to store the input graph topology. In this format, indices of the edges connected to each vertex are stored contiguously in an array, which is denoted as EdgeInfo (EI) in this paper. The offsets to this array are stored in a separate array denoted as VertexInfo (VI). Specifically, the indices of the edges connected to vertex v are stored in EI within the semi-open range $[VI[v], VI[v+1])$. In addition, application specific data structures can be defined per vertex and edge, which are denoted as Vertex Data (VD) and EdgeData (ED) in this paper. As explained in Section 4.5, the Active List (AL) also needs to be stored in main memory.

In the proposed architecture, we define a custom cache corresponding to each graph object type as shown in Figure 2. The access patterns for different object types can vary significantly. For example, EI accesses tend to have good spatial locality because of contiguous storage of indices. On the other hand, VD and ED accesses typically have poor temporal and spatial locality for unstructured graphs due to the random nature of accesses to neighbors' data. The individual cache parameters are customizable in our templatized architecture, and they can be determined based on the specific application requirements.

H. Multiple Accelerator Units

As described before, a single accelerator unit (AU) can process hundreds of vertices and edges concurrently. However, the throughput can be improved further by replicating AUs as shown in Figure 4. In this paper, we focus on fine-grain

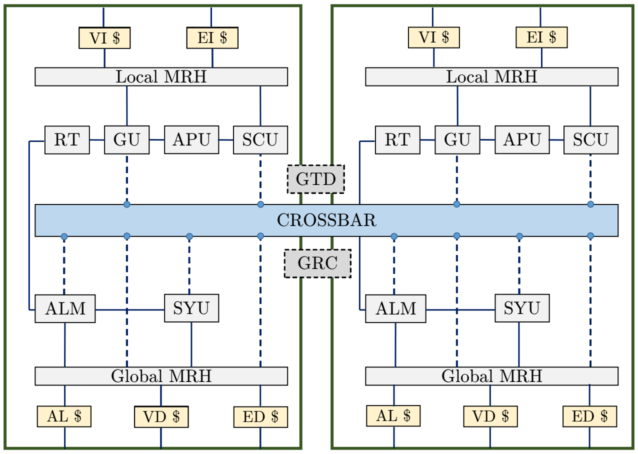

Figure 4: Multiple Accelerator Units with a crossbar

parallelism by tightly integrating a small number of AUs and statically assigning vertices and edges to AUs based on their indices. The memory subsystem is also partitioned according to this assignment in a multi-bank fashion. For k AUs, the partitioning is done based on the bits $[8..8+\log_2 k)$ of the vertex and edge index values.

Assume vertex v and edge e are assigned to AU_v and AU_e, respectively. In this context, we distinguish local and global memory accesses as follows. If a data object associated with vertex v (edge e) can be accessed by only AU_v (AU_e), we define it as a *local* data structure. VertexInfo (VI) and EdgeInfo (EI) are such data structures, and they are accessed through the Local Memory Request Handler (MRH) as shown in Figure 4. Similarly, if a data object can be accessed by multiple AUs, it needs to be accessed through Global MRH of the corresponding AU. VertexData (VD), EdgeData (ED), and ActiveList (AL) are such data structures. Since SYU and ALM modules control memory accesses for VD and AL, respectively, they are connected to the corresponding blocks of the Global MRH.

For example if the GU in AU 0 needs to access the data of a neighboring vertex that is assigned to bank 1, the request is sent to the SYU in AU 1. This SYU checks for RAW hazards, and then forwards the request to the Global MRH in the same AU. When the data response is ready, it is sent back to the GU in AU 0 through the crossbar network.

Although not shown in Figure 4 for clarity, all data caches are connected to a single memory interface for the system DRAM, in a similar way as shown in Figure 2.

When multiple AUs are concurrently running, additional synchronization mechanisms are needed. There are two lightweight modules with minimal processing requirements as outlined below.

Global Rank Counter (GRC): As described in Section IV-D, sequential consistency is implemented by assigning monotonically increasing unique ranks to vertices. When multiple AUs are involved, monotonicity is achieved by a global rank counter (GRC) that sends an increment signal to all SYUs whenever an SYU assigns a new rank. The uniqueness of ranks is ensured by concatenating the AU ID to the least significant bit of the original ranks. Although not shown in Figure 4, GRC is connected to the SYU of each AU.

Table I: Parameters used for Accelerators Constructed

		Gather Unit		Scatter Unit		Cache
	# AUs	# vtxs	# edges	# vtxs	# edges	size
PR	4	32	128	16	128	9.9 KB
SSSP	4	32	4	16	128	8.9 KB
LBP	4	16	64	16	64	34.8 KB
SGD	4	16	64	16	64	9.6 KB

Global Termination Detector (GTD): The Runtime (RT) of each AU is responsible for detecting termination condition for that AU. When multiple AUs are involved, GTD collects the termination signals from individual RTs, and determines the termination condition of the whole system. GTD is responsible for notifying the host processor that the computation is finished.

Note that GRC and GTD are the only centralized modules in a multi-AU system. Both implement very simple operations that are not in the critical path for performance. Hence, the execution happens in a distributed fashion without any centralized bottleneck.

V. Experimental Setup

Using the proposed architecture template, we generated accelerators for 4 applications (outlined in Section V-A), and compared with a state-of-the-art IvyBridge server system. Details of the execution environments are as follows:

- *CPU:* This is the baseline against which we compare our accelerators. The system is composed of two sockets. Each socket has 12 cores. Each core has private L1 and L2 caches, and the L3 cache is shared by cores on the same socket. Total cache capacity is 768KB, 3MB and 30MB for L1, L2 and L3 respectively. Total DRAM capacity of the system is 132GB. Software is implemented in OpenMP/C++. Applications are either hand optimized, or reused from existing benchmark suites. Each application is compiled using gcc 4.9.1 version with -O3 flag enabled. When needed, we set the NUMA policy to divide the memory allocation for an application to two different sockets on the system to maximize the memory bandwidth utilization. The applications in our experiments cannot effectively utilize the vector extensions of the CPU due to the reasons explained in Section II-B.
- *ACC:* This is the accelerator generated by the proposed architecture template for each application. The architectural parameters are customized per application. The main parameters are listed in Table I. Observe that 4 Accelerator Units (AUs) are used for all applications. The number of vertices and edges concurrently processed in each AU are also listed for the Gather and Scatter Units. Finally, the total cache storage in the memory subsystem of each AU is listed in the last column.

As discussed in Section II-B, GPUs are not well-suited for irregular graph applications. There are several existing works that have compared GPU performance with CPUs. For example, it is reported that a GPU implementation of Stochastic Gradient Descent (SGD) performs as good as 14 cores on a 40-core CPU system [20]. For Single-Source Shortest Path (SSSP) problem, it is reported that an efficient serial implementation can outperform highly parallel GPU implementations for high-diameter or scale-free graphs [21]. A GPU-based sparse matrix-vector multiplication implementation of PageRank has been proposed recently [22], where 5x speed-up is observed with respect to a 4-core CPU. However, this work ignores the work-efficiency advantages of asynchronous execution and asymmetric convergence (Section II). It has been shown that a synchronous implementation can be up to 3x less work efficient compared to an implementation that keeps track of active vertices and performs asynchronous computation [3].

A. Graph Applications

In order to test our framework, we have selected widely used graph applications from different domains, such as machine learning, computer vision, and data mining, which are briefly described below.

PageRank (PR): It is an important graph application used to order web pages according to their importance. The pseudo code of the algorithm is given in Figure 1. As a baseline, we use the multi-core CPU implementation from the Berkeley GAP Benchmarks [23]. We extended the existing implementation to improve convergence behavior by adding a bit vector to keep track of active vertices. In our ACC implementation, the *PageRank (PR)* value of a vertex and $1/out\_degree$ of a vertex is stored as vertex data. In a vertex program execution, the current vertex collects and accumulates the *PR* values from its neighbors and updates its own *PR* value.

Loopy Belief Propagation (LBP): It is a well-known image stitching algorithm that works on a grid graph. Each vertex in the graph represents a pixel of a given image. More specifically, each vertex has a belief vector where each entry represents the probability of the corresponding label for the vertex. The following 3 stages are performed for each vertex: 1) The messages from neighbors are accumulated. 2) The belief of the vertex is updated based on the accumulated value. 3) A new message is generated using min convolution and sent to each neighboring vertex. The CPU implementation uses a synchronized execution model. Specifically, a ”2-coloring” scheme is implemented, where the vertices with the same color are executed in parallel, avoiding the need for locks. Our ACC implementation is similar to GraphLab, where edges store the messages in both directions and vertices keep belief values. Initially, a vertex program visits all incident edges of the vertex and calculates the log-sum (product) of all incoming messages. Then, the vertex updates the belief value of its own. Finally, for each incident edge of this vertex, the outgoing message values are updated.

Stochastic Gradient Descent (SGD): It is an iterative machine learning algorithm used in recommender systems. SGD operates on a bipartite graph, and tries to estimate a feature vector for both user and item vertices of the graph. The dot product of a user and item vector is expected to give the estimated rating of the user for that item. We used the DSGD algorithm [24] as the baseline CPU implementation because it is shown to be the most efficient implementation of SGD in [19].

Single Source Shortest Path (SSSP) As a baseline CPU implementation, we used the SSSP implementation from the Berkeley GAP benchmarks [23]. Special bucket-based data structures are used in that implementation to achieve high parallel performance. In our accelerator implementation, distance of a vertex to a source node is stored as vertex data. If the distance value of a vertex is updated, it sends a message to its neighbors. Each edge is assigned a unit weight in the input data that we use in our experiments.

B. Power, Performance, and Area Estimation

1) Methodology for CPU

We used the time measurement function calls that exist in the OpenMP library. To calculate the energy and power consumption of the native system, we used Running Average Power Limit (RAPL) [25], which provides energy measurements for core, uncore and DRAM by allowing us to read MSR registers. The baseline CPU system uses DDR3 as the system memory. For fair power comparisons with accelerators, we estimated DDR4 power consumption separately from the CPU system with DRAMSim2 [26]. For that purpose, we generated DDR4 access traces that result in the same bandwidth as DDR3 of the CPU system and then applied them to DRAMSim2 with a DDR4 device model.

2) Methodology for Accelerator Compute Blocks

We used a commercial high-level synthesis (HLS) tool to generate RTL from our SystemC-based performance models in order to estimate area, performance and power for each block. HLS was run for each application on five main blocks of the accelerator unit: Gather, Scatter, Apply, Sync and ALM. Then, the generated RTL was run through a commercial physical-aware logic synthesis tool to confirm absence of timing violations and to measure area and power at the gate-level. We used 22*nm* technology library for standard cells and metal layers and a 1GHz clock frequency. Significantly large arrays (about 1Kb and larger) were implemented using synthesizable latch-based register files (RF). For the crossbar block, we estimated the wire length of interconnects between all the blocks in the accelerator unit based on an approximate floorplan and the area of the blocks. We estimated area and power for the crossbar using the wire length, routing resources and physical parameters of the metal layers.

Most of the SystemC template-related functions were modeled at cycle-accurate level to provide accurate performance estimates. Application-specific functions, such as scatter, gather and apply, were pipelined using the pipelining feature of the HLS tool. We were able to achieve a throughput of 1 function call per cycle for every user function except the Gather function of SGD, which has throughput of 1/4 due to

Table II: Datasets used in our experiments.

Application	Dataset	# Vert.	# Edges
PageRank SSSP (Directed)	wg	916K	5.1M
	pk	1.6M	30M
	lj	4.8M	69M
	g24	16.8M	268M
	g25	33.5M	536M
	g26	67M	1000M
LBP (Undirected)	1M	1M	2M
	4M	4M	8M
	9M	9M	18M
SGD (Undirected)	1M	9.7K	1M
	10M	80K	10M

significantly more computation done per vertex. The latencies were also one except for the *Apply* function of PageRank, *Scatter* function of LBP, and the *Gather* function of SGD. Their latencies are 3, 6, and 4 cycles, respectively.

The latency/throughput values for user functions are back-annotated to the original SystemC model for performance measurements. For power measurements, we used a hybrid SystemC-Verilog simulation methodology, where RTL for the block of interest and annotated SystemC models for the rest of the blocks were used to generate power traces. During simulation, we captured switching activity for all inputs and sequential elements of the RTL block in SAIF format. Then, we used a commercial power analysis tool that takes the SAIF file as input and produces power values for the given switching activity file.

3) Methodology for Memory Subsystem

The accelerator memory subsystem is composed of internal memories such as caches and light-weight load/store queues, and DRAM. We estimate the power and area of internal memories using Cacti 6.5 [27]. Since Cacti only supports down to 32*nm* technology, we apply three different scale factors to convert them to 22*nm* technology. For area, we used the scaling factor 0.5 based on [28, 29]. For dynamic power, we used the scaling factor 0.569 as in [30]. Finally, for the leakage power, we used the scaling factor 0.8 as in [31]. In order to estimate dynamic power consumption, we first compute the dynamic energy consumption by measuring access count of each memory component and then multiplying it by the energy per access provided by Cacti. For example, we collect energy per access through Cacti for a cache and run simulation to get the access count of the cache. Then, we multiply them together to estimate the dynamic energy consumption of the cache. For leakage energy, since leakage current is always consumed as long as power is turned on, we simply multiply the total execution time by leakage power. By summing up the dynamic and leakage energy, we can compute the total energy consumption. Total power consumption is simply computed by dividing total energy with total execution time. DRAM power is computed using DRAMSim2[26] with a DDR4 memory model.

C. Datasets

We tested each application with several datasets, either taken from existing graph datasets or created synthetically. For SSSP and PageRank applications, we selected 3 different directed graphs from the SNAP datasets [32]: *WebGoogle(wg)*, *soc-Pokec(pk)*, and *soc-LiveJournal(lj)*. We generated three large graphs using Graph500 [33] with 16, 32, and 67 millions vertices. For LBP, we synthetically generated three different graphs using GraphLab's synthetic image generator [1]. Each image has 4 different colors, and hence, there are 4 different possible labels for each pixel. Images generated for LBP tests include 1000x1000, 2000x2000, and 3000x3000 pixels (vertices). For the SGD application, we selected two different movie datasets from MovieLens [34]. The first movie dataset (1M) includes approximately 1 million ratings and the second one (10M) includes approximately 10 million ratings. Table II shows the detailed description of each dataset with its respective properties.

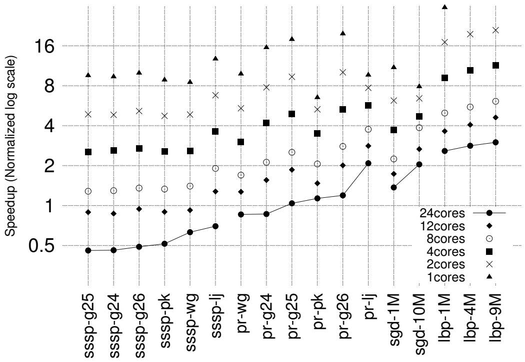

Figure 5: Execution time comparisons. The y-axis is the speed-up of the proposed accelerators with respect to multi-core execution.

VI. Experimental Results

In this section, we present our experimental results in terms of execution time, power and area. We provide results for 17 different test cases, where each test case is an application-dataset pair.

A. CPU and Accelerator Comparison

1) Execution Time and Throughput

As mentioned in Section V, we used a 24-core server system as our baseline for these experiments. We used identical convergence conditions for the CPU and accelerator implementations so that the execution time comparisons make sense. In this section, we report the performance results in terms of total execution times (Figure 5) and throughput values (Figure 6). Throughput is defined as the number of edges processed per second. Note that throughput is a raw performance metric, because it does not take into account the convergence behavior. As shown in [3], an implementation can have higher throughput, but worse execution time, especially if the properties described in Section 2 are not taken into account.

PageRank is one of the best examples of an iterative, converging graph application, which benefits from asynchronous execution as shown in [1]. Although the baseline CPU

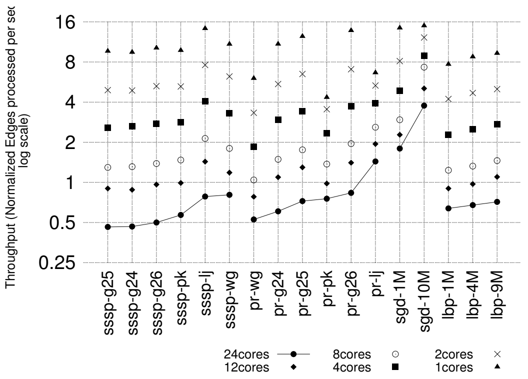

Figure 6: Throughput comparisons. The y-axis is the ACC throughput divided by the CPU throughput.

implementation has some asynchronous execution support, its vertex scheduling is not asynchronous [3]. Observe that the performance benefits of our accelerators are higher when the execution time metric is considered (Figure 5) compared to the raw throughput metric alone (Figure 6). This shows the importance of the asynchronous mode support in our architectures. Compared to the 24-core system, our accelerators have better or equivalent execution times in 4 out of 6 test cases[4]. Compared to 12 or fewer cores, the speed-up observed is in the range of 2x to 20x.

For the LBP application, observe that our throughput values are comparable to the throughput of 12 cores. However, when the total execution time is considered, our accelerator is between 2.5x and 3x faster than 24-cores. We believe the reason for this is the sequential consistency support provided in our accelerators. It was shown in [1] that LBP-like applications have much better convergence behavior when sequential consistency is enabled. However, as shown in [3], implementing sequential consistency on a CPU can slow down the execution by up to an order of magnitude due to extra locking overheads.

For SGD, our accelerators perform better than a 24-core CPU in terms of both execution time and throughput metrics. The reason is the large number of arithmetic operations performed per vertex, which is done more efficiently with custom hardware.

SSSP is the only application where our accelerators do not outperform 24-core performance. The baseline CPU implementation is highly optimized with special data structures that cannot be modeled as a vertex-centric program alone. As future work, such data structures can be added to our accelerator templates. The performance of our accelerators is similar to the performance of 12-core CPU. However, as will be shown in Section VI-A2, our accelerators consume significantly less power than 12 cores.

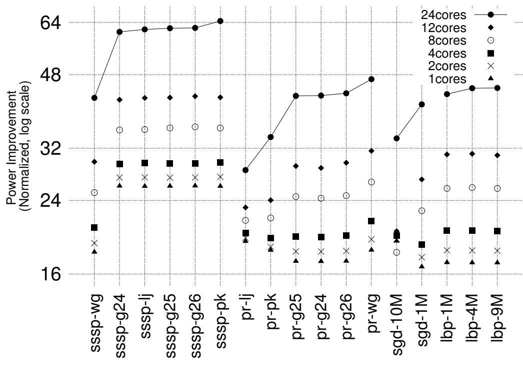

Figure 7: Power consumption comparisons. The y-axis is the CPU power divided by the ACC power.

2) Power Efficiency

Power consumption of our accelerators is dominated by the DDR4 power, which is around 3W for all test cases. This is about 8x larger than the power consumed by the rest of the system, including all accelerator units and cache structures. Other studies have also observed that accelerator power is dominated by DRAM access [35]. However, for CPU executions, core+uncore power consumption is much larger than the projected DDR4 power values.

Figure 7 shows the power consumption of the baseline CPU with respect to our accelerators. Note that the CPU power includes core, uncore, and the projected DDR4 power values. The accelerator power includes all accelerator units, caches, and DDR4. Observe that our accelerators have up to 65x better power efficiency compared to the CPU system. Most importantly, even if our SSSP accelerator does not perform as fast as 24 cores (Section VI-A1), we observe about 64x lower power for most of the SSSP test cases.

B. Area and Power Analysis of Accelerator

In this subsection, we provide the detailed power and area breakdown of accelerator units and cache units. As stated in Section V-A, different applications have different computational requirements. As shown in Tables III and IV, area and power consumption of individual blocks depend on the application. For example, for PageRank and SGD the *gather unit* occupies the most area while the *scatter unit* takes most of the area for LBP. Beside computational units, depending on the application requirements, different basic blocks in the accelerator unit can occupy different areas. For example, LBP and SGD both implement support for *sequential consistency*. Their synchronization unit occupies a larger area than PageRank and SSSP.

Beside computational units, cache components also depend on the data structures that are used in the application definition (see Section IV-G for the acronyms used for different data structures). Tables V and VI show the details for each cache unit. The data structure that has the maximum amount of storage has generally the highest amount of power consumption and area. For example, when we consider the

[3] Fully asynchronous multi-core implementation would require more synchronization, which would lead to worse execution times.

[4] The remaining two test cases are smaller, and CPU has better LLC utilization for these cases. We would also expect better performance if our accelerators were connected to an LLC instead of directly to DRAM.

Table III: Power Breakdown of Accelerator Units(in W)

	Pagerank		LBP		SGD		SSSP	
	Power	%	Power	%	Power	%	Power	%
gather	0.029	33	0.045	23	0.438	80	0.008	13
scatter	0.015	16	0.066	34	0.012	2	0.022	39
apply	0.011	12	0.006	3	0.007	1	0.001	2
sync	0.014	16	0.035	18	0.062	11	0.007	13
alm	0.013	15	0.014	7	0.014	3	0.012	22
runtime	0.000	1	0.001	0	0.000	0	0.001	1
crossbar	0.006	7	0.029	15	0.013	2	0.006	10

Table IV: Area Breakdown of Accelerator Units (in mm^2)

	Pagerank		LBP		SGD		SSSP	
	Area	%	Area	%	Area	%	Area	%
gather	0.238	54	0.192	25	0.484	42	0.090	31
scatter	0.096	22	0.247	32	0.101	9	0.121	42
apply	0.030	7	0.010	1	0.012	1	0.005	2
sync	0.032	7	0.244	31	0.504	43	0.032	11
alm	0.030	7	0.029	4	0.029	3	0.029	10
runtime	0.002	0	0.002	0	0.002	0	0.002	0
crossbar	0.011	3	0.051	7	0.024	2	0.010	4

Table V: Power Breakdown for Cache structures (in W).

	Pagerank		LBP		SGD		SSSP	
	Power	%	Power	%	Power	%	Power	%
VI	0.0019	6	0.0028	4	0.0007	2	0.0015	5
EI	0.0021	7	0.0080	13	0.0038	12	0.0008	3
VD	0.0098	33	0.0108	18	0.0069	23	0.0007	3
ED	0.0000	0	0.0199	33	0.0000	0	0.0000	0
AL	0.0160	54	0.0141	24	0.0191	63	0.0245	89
L/S Unit	0.0000	0	0.0044	7	0.0000	0	0.0000	0

Table VI: Area Breakdown for Cache Structures (in mm^2).

	Pagerank		LBP		SGD		SSSP	
	Area	%	Area	%	Area	%	Area	%
VI	0.0105	9	0.0187	5	0.0077	6	0.0082	9
EI	0.0050	4	0.0473	14	0.0095	8	0.0026	3
VD	0.0175	15	0.0647	19	0.0167	14	0.0037	4
ED	0.0000	0	0.1054	30	0.0000	0	0.0000	0
AL	0.0822	72	0.0830	24	0.0849	72	0.0799	84
L/S Unit	0.0000	0	0.0288	8	0.0000	0	0.0000	0

PageRank application, vertex data (VD) is the only storage that the application has and we observe that 32% of power consumption belongs to this cache. The same characteristics are also valid for other applications such as LBP and its edge data (ED) cache, SGD and its VD cache. In addition to caches that are used for the application data storage, active list (AL) caches consume significant amount of power and area in our accelerator architecture. Yet, the power consumption of the memory subsystem is still negligible compare to the 3W DRAM power.

C. Scalability and Sensitivity Analysis

The default architecture parameters for the proposed accelerators are listed Table I. In this section, we change one parameter at a time and measure the change in performance.

As described in Section IV, processing multiple vertices and edges allows us to achieve high levels of memory level parallelism and tolerate long latencies. Figure 8 illustrates the performance sensitivity with respect to the number of concurrent edges in Gather and Scatter Units of a single AU. Here, the y-axis value of 1.0 corresponds to the execution time for the parameters in Table I, and values larger than 1.0 correspond to slower executions due to parameter change. Observe that a certain number of concurrent vertices and edges are needed to achieve the best performance, after which the performance saturates. This is due to Little's Law, which states that the number of in-flight requests need to be at least throughput times latency to be able to fully utilize the available DRAM bandwidth.

VII. Related Work

Previous work can be categorized into three main categories. There have been several proposals for graph processing environments to efficiently execute graph applications. One of the first is Google's Pregel [14]. Pregel suggests a bulk synchronous environment which avoids the usage of locks and focuses on very large scale computing. On the other hand, GraphLab [1] focuses on asynchronous computations and benefits from convergence characteristics of applications. Galois [18] is another framework which also gives better performance compared to naive implementations of applications. Other examples of software solutions for graph applications are Giraph [15], CombBLAS [16], and SociLite [17]. While these are all optimized for graph parallel applications, they are purely software-based systems. Our approach can be extended to support any of these frameworks.

Secondly, there have been efforts on accelerating graph applications. In [36], the authors propose a warp centric execution model for graph applications. Additionally, Medusa [37] is a processing framework which focuses on bulk synchronous processing and targeted for GPUs. They also consider multi GPU acceleration and optimize graph partitioning to reduce the communication between GPUs. [38] adapts vertex centric and message passing execution for CPU and MIC.

Finally, there are existing works on architectural support for graph applications. One of these approaches [39] tries to implement a hardware work-list that would make data driven executions for irregular applications feasible on GPGPUs. On the other hand, GraphGen [40] is a framework to create application specific synthesized graph processor and memory layout for FPGAs. GraphGen also uses a vertex centric execution model to represent graph applications. However, it is targeted towards regular applications and cannot handle irregular applications such as PageRank. Recently, [5] has provided a PIM (processing in memory) system that uses 3D integration technology, and tries to maximize the available memory bandwidth. GraphStep [41] implements a bulk synchronous message passing execution model on FPGAs for graph applications. To the best of our knowledge, our work is the first accelerator architecture that specifically targets asynchronous, iterative, vertex-centric graph applications with irregular access patterns and asymmetric convergence.

VIII. Conclusions and Future Work

The main contributions of this paper are: 1) an accelerator architecture for iterative vertex-centric graph applications with irregular access patterns and asymmetric convergence, 2) a hardware template to model graph analytics applications, and 3) a detailed experimental study through physical-aware

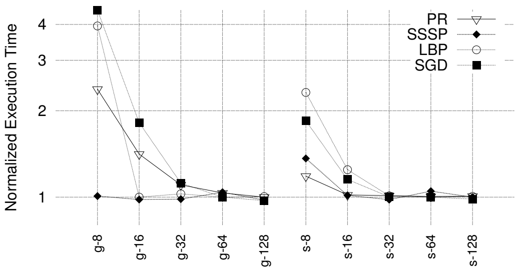

Figure 8: Sensitivity analysis for the number of concurrent edges in (g-XX) Gather Unit and (s-XX) Scatter Unit of a single AU. (XX is the number of concurrent edges in the corresponding unit)

RTL synthesis using industrial 22nm libraries. Our proposed accelerators have performance similar as or better than a state-of-the-art 24-core CPU system for most of our test cases, sometimes outperforming the 24-core system by up to 3x. More importantly, we have estimated the area requirement and power consumption of these hardware accelerators through physical-aware synthesis and show that significant improvements can be achieved both in terms of area and power. The results have shown that our proposed accelerator is more power efficient than the 24-core server system by up to a factor of 65x. Furthermore, the aforementioned improvements are obtained with two orders of magnitude smaller area requirements.

Although this paper has focused on fixed function accelerators, the proposed architectural features are also applicable to programmable hardware. In particular, we are currently working on FPGA implementation of the proposed template. Another planned future work is to replace the application-specific logic with simple processors to create software programmable graph accelerators.

References

[1] 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.

[2] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, “Distributed graphlab: A framework for machine learning in the cloud,” *CoRR*, vol. abs/1204.6078, 2012.

[3] M. Ozdal, S. Yesil, T. Kim, A. Ayupov, S. Burns, and O. Ozturk, “Architectural requirements for energy efficient execution of graph analytics applications,” in *Proc. of International Conference on Computer-Aided Design (ICCAD)*, 2015.

[4] A. Arasu, J. Novak, A. Tomkins, and J. Tomlin, “Pagerank computation and the structure of the web: Experiments and algorithms,” in *Proc. of the Eleventh International World Wide Web Conference*, 2002.

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *Proc. of the 42Nd Annual International Symposium on Computer Architecture*, ISCA ’15, (New York, NY, USA), pp. 105–117, ACM, 2015.

[6] R. Albert and A.-L. Barabási, “Statistical mechanics of complex networks,” *Reviews of Modern Physics*, vol. 74, pp. 47–97, Jan. 2002.

[7] S. Suri and S. Vassilvitskii, “Counting triangles and the curse of the last reducer,” in *Proc. of the 20th International Conference on World Wide Web*, WWW ’11, (New York, NY, USA), pp. 607–614, ACM, 2011.

[8] S. Beamer, K. Asanovic, and D. Patterson, “Locality exists in graph processing: Workload characterization on an ivy bridge server,” in *Proc. of IEEE International Symposium on Workload Characterization (IISWC)*, pp. 56–65, October 2015.

[9] M. Garland and D. B. Kirk, “Understanding throughput-oriented architectures,” *Communications of the ACM*, vol. 53, no. 11, pp. 58–66, 2010.

[10] Q. Xu, H. Jeon, and M. Annavaram, “Graph processing on gpus: Where are the bottlenecks?,” in *Proc. of IEEE International Symposium on Workload Characterization (IISWC)*, pp. 140–149, October 2014.

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

[12] N. Chatterjee, M. O’Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, “Managing dram latency divergence in irregular gpgpu applications,” in *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 128–139, November 2014.

[13] M. O’Neil and M. Burtscher, “Microarchitectural performance characterization of irregular gpu kernels,” in *Proc. of IEEE International Symposium on Workload Characterization (IISWC)*, pp. 130–139, October 2014.

[14] 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 *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[15] “Giraph.” http://giraph.apache.org/.

[16] “The combinatorial BLAS: Design, implementation, and applications.” http://gauss.cs.ucsb.edu/ aydin/combblas-r2.pdf.

[17] M. S. Lam, S. Guo, and J. Seo, “Socialite: Datalog extensions for efficient social network analysis,” in *Proc. of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE ’13, (Washington, DC, USA), pp. 278–289, IEEE Computer Society, 2013.

[18] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, “The tao of parallelism in algorithms,” *SIGPLAN Not.*, vol. 46, June 2011.

[19] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, “Navigating the maze of graph analytics frameworks using massive graph datasets,” in *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’14, (New York, NY, USA), pp. 979–990, ACM, 2014.

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

[21] A. Davidson, S. Baxter, M. Garland, and J. Owens, “Work-efficient parallel gpu methods for single-source shortest paths,” in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 349–359, May 2014.

[22] X. Yang, S. Parthasarathy, and P. Sadayappan, “Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining,” *Proc. VLDB Endow.*, vol. 4, pp. 231–242, Jan. 2011.

[23] “Gap benchmark suite code.” https://github.com/sbeamer/gapbs.

[24] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, “Large-scale matrix factorization with distributed stochastic gradient descent,” in *Proc. of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’11, (New York, NY, USA), pp. 69–77, ACM, 2011.

[25] “Running average power limit.” https://01.org/blogs/tlcounts/2014/running-average-power-limit-%E2%80%93-rapl.

[26] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, “DRAMSim2: A cycle accurate memory system simulator,” *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, 2011.

[27] “Cacti.” http://www.hpl.hp.com/research/cacti.

[28] M. Bohr, “Silicon technology leadership for the mobility era,” in *IDF*, 2012.

[29] M. Bohr, “14nm process technology: Opening new horizons,” in *IDF*, 2014.

[30] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron, “Scaling with design constraints: Predicting the future of big chips,” in *IEEE Micro*, 2011.

[31] O. Villa, D. R. Johnson, M. O’Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally, “Scaling the power wall: A path to exascale,” in *Proc. of Supercomputing*, 2014.

[32] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection.” http://snap.stanford.edu/data, June 2014.

[33] “Graph500.” www.graph500.org.

[34] “Movielens dataset.” http://grouplens.org/datasets/movielens/.

[35] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” *SIGARCH Comput. Archit. News*, vol. 42, pp. 269–284, Feb. 2014.

[36] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, “Accelerating CUDA graph algorithms at maximum warp,” *SIGPLAN Not.*, vol. 46, pp. 267–276, Feb. 2011.

[37] J. Zhong and B. He, “Medusa: A parallel graph processing system on graphics processors,” *SIGMOD Rec.*, vol. 43, pp. 35–40, Dec. 2014.

[38] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Efficient and simplified parallel graph processing over CPU and MIC,”

[39] J. Y. Kim and C. Batten, “Accelerating irregular algorithms on GPGPUs using fine-grain hardware worklists,” in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 75–87, Dec 2014.

[40] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martinez, and C. Guestrin, “Graphgen: An FPGA framework for vertex-centric graph computation,” in *Proc. of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, FCCM ’14, (Washington, DC, USA), pp. 25–28, IEEE Computer Society, 2014.

[41] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. Uribe, J. Knight, T.F., and A. DeHon, “Graphstep: A system architecture for sparse-graph algorithms,” in *Field-Programmable Custom Computing Machines, 2006. FCCM ’06. 14th Annual IEEE Symposium on*, pp. 143–151, April 2006.