Graph Analytics Accelerators for Cognitive Systems

Muhammet Mustafa Ozdal†, Serif Yesil‡, Taemin Kim*, Andrey Ayupov*, John Greth*, Steven Burns*, and Ozcan Ozturk†
† {mustafa.ozdal, 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
‡ syesil2@illinois.edu
Univ. of Illinois at Urbana-Champaign Urbana, IL 61801

Abstract—Hardware accelerators are known to be performance and power efficient. In this paper, we focus on accelerator design for graph analytics applications which are commonly used kernels for cognitive systems. We propose a templatized architecture that is specifically optimized for vertex centric graph applications with irregular memory access patterns, asynchronous execution, and asymmetric convergence. Proposed architecture addresses the limitations of the existing CPU and GPU systems while providing a customizable template. Our experiments show that the generated accelerators can outperform a high end CPU system with up to 3x better performance and 65x better power efficiency.

Index Terms—C.1.4 Parallel Architectures, C.3 Special-Purpose and Application-Based Systems,

1 Introduction

Cognitive systems consist of many elements such as natural language processing, artificial intelligence, machine learning, and data analytics [1]. These systems rely on processing large amounts of data, hence there are ongoing efforts to integrate big data analytics with cognitive systems [2]. Graph analytics has been gaining popularity recently especially due to the abundance of data from web and social networks. Specifically, graph-based large knowledge bases can be used by cognitive systems for reasoning and human interactions [3].

There are many graph algorithms that are executed in the inner loops of cognitive systems. Many speech recognition and language processing models used in cognitive applications can be represented as graphs [4]. For example, finite-state decoding graphs are commonly used by WFST-based speech recognition algorithms, where the objective is to compute the most likely paths corresponding to the input sequences [4]. *Belief propagation* on a Bayesian network is another graph analytics application that is used by different cognitive applications [2]. Cognitive platforms such as Unmanned Aerial Vehicles (UAVs) or self-driving cars run single-source shortest-path (SSSP) algorithms in their inner loop computations [5]. Personalized recommendations can be generated from large datasets using algorithms such as Stochastic Gradient Descent (SGD) on bipartite graphs. TextRank is a model proposed for natural language applications such as tagging documents with key phrases and sentence extraction for automatic summarization [6]. TextRank operates on graphs where vertices represent text units (e.g. terms) and edges represent the associations inferred from the input texts.

The preliminary version of this article was published in ACM/IEEE International Symposium on Computer Architecture (ISCA) 2016.

Graph analytics-based cognitive applications are different than traditional compute-intensive applications with regular access patterns and abundant data and thread level parallelism. As will be discussed later, graph applications are hard to parallelize due to irregular execution patterns and synchronization requirements. In this paper, we propose an accelerator architecture that is specifically targeted at graph analytics applications. The proposed architecture is implemented as a template to make modeling different applications easy. Architects and designers can plug in application-level data structures and functions into this template to generate hardware implementations for a large class of graph analytics applications.

2 Graph Analytics Applications

Graph analytics applications are among the core algorithms used in cognitive systems. However, implementing these applications on existing systems efficiently is not a trivial task due to several reasons such as memory access bottlenecks, synchronization problems, and irregular computation/communication patterns. As will be discussed shortly, these properties can be exploited to improve performance and power efficiency by hardware customization.

Many graph algorithms are iterative in nature, where execution continues until a convergence criteria is met. However, the number of iterations required to converge individual vertices can vary significantly. For instance, Figure 1(a) plots the percent of vertices not converged throughout the PageRank iterations, where only *less than 1%* of vertices require all iterations to converge. Our analysis shows that enabling asymmetric convergence decreases the total number of edges processed by 47% on average.

While single instruction multiple data (SIMD) type of architectures can process multiple vertices simultaneously, they

have control divergence issues when asymmetric convergence is enabled.

Asynchronous execution, which allows neighbor vertices to access the most recent data, can improve work efficiency even further [7]. Figure 1(b) shows the relative number of edges processed for PageRank when support for both asynchronous execution and asymmetric convergence is enabled. As shown here, these features reduce the number of edges processed by 66% on average. However, if a programmer wants to utilize the work efficiency of asynchronous execution then he/she should handle synchronization and enforce sequential consistency in software to prevent data race conditions. Fine-grain locking mechanisms slow down the execution of both CPUs and throughput architectures.

One of the main bottlenecks in graph applications is memory access because of low compute-to-communication ratios, low spatial/temporal locality, and hard-to-predict memory accesses.

Some real world graphs, such as social networks, follow Power law distribution where a few vertices have much higher degrees when compared to the rest. In such an environment, static partitioning of vertices to different threads greatly suffer from load imbalances. An efficient implementation needs to distribute its workload carefully, and more importantly, it should be able to handle high degree vertices efficiently.

3 Proposed Architecture Template

There have been several graph frameworks proposed in the last few years. The common objective is to hide the complexities of parallel and distributed software development by providing high-level programming interface. We follow a similar approach for our template architecture. Specifically, we use the vertex-centric ("think like a vertex") abstraction model that consists of Gather-Apply-Scatter (GAS) functions as in GraphLab [7]. In this model, 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 application specific data structures and functions in the programming interface are clearly separated from the architecture template implementation. 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 40 lines of C code, while the common architecture template is more than 30,000 lines of SystemC code, and not visible to the user.

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(a) illustrates the proposed high-level architecture for a single accelerator unit (AU). The 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.
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.
3) Synchronization between concurrently processed vertices and edges is done in the Sync Unit (SYU) module, which is specifically designed for graph processing. This module ensures sequential consistency with negligible performance overhead. Furthermore, it works in a distributed fashion without a centralized bottleneck.
4) The set of active (not-yet-converged) vertices is maintained by the Active List Manager (ALM) module. 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.

In the following subsections, we describe different modules in a single AU and then explain how to connect multiple AUs together.

3.1 Computational Units

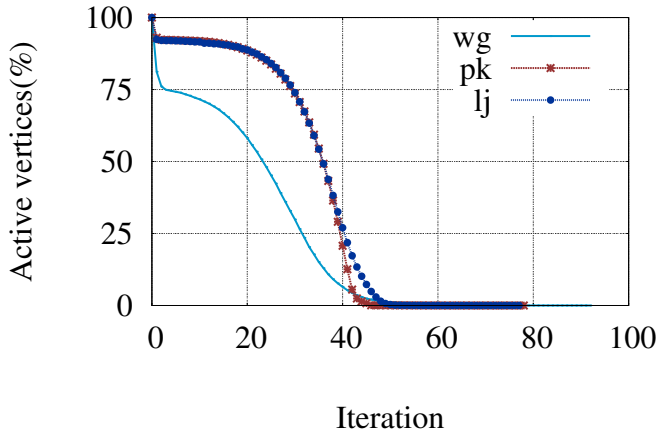
The main computational units in our accelerator are *Gather Unit (GU)*, *Apply Unit (APU)*, and *Scatter Unit (SCU)* as shown in Figure 2(a). These computational units are designed to perform the respective Gather, Apply, and Scatter operations for each vertex.

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 vertex and edge states are stored locally.

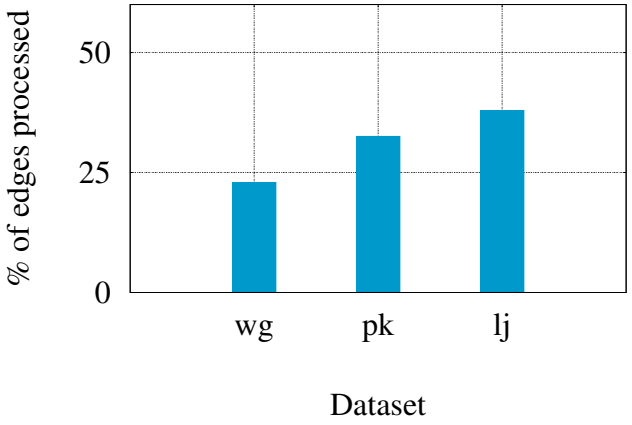
The limited local storage available in GU is shared among all concurrently processed vertices. In the GU microarchitecture we propose, a credit based mechanism is used to assign the available edge slots dynamically to multiple vertices. The vertices that are supposed to execute logically before others 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.

The APU is the module that performs computation for each vertex using the data computed by GU without accessing 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.

SCU implements the Scatter Program for each vertex v, where 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.

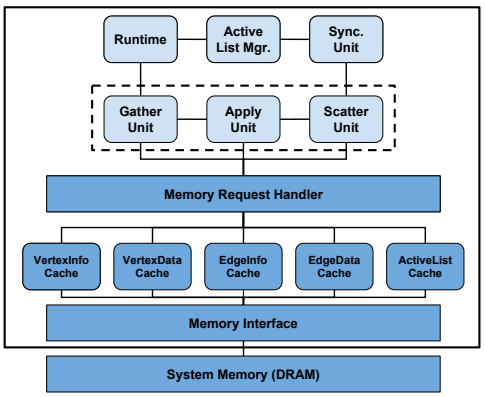


(a) Asymmetric convergence of vertices for PageRank.

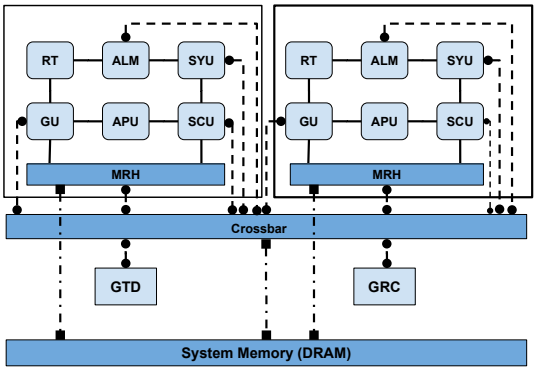


(b) Work efficiency when asymmetric convergence and async. execution is enabled.

Fig. 1: Analysis of the PageRank application on three datasets: wg, pk, and lj.



(a) Single accelerator unit.



(b) Multiple accelerator units.

Fig. 2: Accelerator block diagram. For clarity some of the connections between the blocks are not shown.

3.2 Enabling Sequential Consistency

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 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 [7], which implements sequential consistency by enforcing ordering between adjacent vertices, since 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 (as will be described). 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. The neighboring vertex data (NVD) access requests from Gather Unit (GU) go through SYU to ensure this ordering.

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 a *message* to SYU corresponding to each edge $e : u \rightarrow v$, and waits for acknowledgement before it writes data associated with e or u.

Avoid Unnecessary Activations: Consider an activation message received from SCU corresponding to edge $u \rightarrow v$. This implies that 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.

3.3 Managing Active Vertices

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.

3.4 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.

3.5 Specialized 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). The offsets to this array are stored in a separate array denoted as VertexInfo (VI). In addition, application specific data structures can be defined per vertex and edge, which are denoted as Vertex Data (VD) and EdgeData (ED). As explained previously, 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(a). 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.

3.6 Multiple Accelerator Units

The throughput can be improved further by replicating AUs as shown in Figure 2(b). In this paper, we focus on fine-grain 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.

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

Global Rank Counter (GRC): As described previously, 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. GRC is connected to the SYU of each AU.

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.

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.

4 Empirical Study

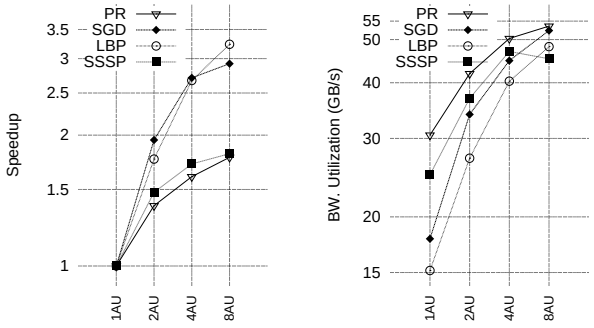
4.1 Setup

In this work, we have selected 4 different graph analytics applications which are used as part of cognitive systems.

PageRank (PR) is a well known ranking algorithm, where it is used not only to rank web pages, but also to summarize text as in TextRank and LexRank [8]. *Stochastic Gradient Descent (SGD)* is an iterative machine learning algorithm which is used in personalized recommendations. This is especially important as many web services depend on personalized recommendations to improve user experience. Additionally, *Single Source Shortest Path (SSSP)* is a kernel used in UAVs which are among future cognitive systems. SSSP is also used in network analytics as a kernel for Betweenness Centrality calculations. Moreover, image processing and belief propagation are crucial to cognitive systems which *Loopy Belief Propagation (LBP)* collectively uses.

TABLE 1: Datasets used in our experiments [9].

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



(a) Execution Time (b) Bandwidth

Fig. 3: Scalability of accelerator in terms of execution time and memory bandwidth utilization normalized with respect to 1 accelerator).

For each benchmark, we tried to use the most efficient implementations either by taking from available benchmark suites or manually optimizing. Further details of applications and selected benchmark implementations can be found in our preliminary work [9].

We test each application with various datasets taken from well-known graph databases [9]. The number of vertices and edges in selected graphs ranges from 916K vertices and 5.1M edges to 67M vertices and 1000M edges for PageRank and SSSP applications. On the other hand, 3 different images are used for LBP with 1000x1000, 2000x2000, and 3000x3000 pixels. For SGD, 2 movie datasets are selected with approximately 1M and 10M ratings. Details of the selected datasets can be found in Table 1.

To calculate the energy and power consumption of the native system, we used Running Average Power Limit. On the other hand, for our accelerator, we used 22nm libraries for standard cells and metal layers, CACTI for caches, and DramSim2 for memory. 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 all applications, the number of AUs is chosen as four. However, other microarchitectural parameters (e.g. cache sizes, number of vertices/edges processed concurrently) have been tuned individually per application. Further details of our experimental setup including the parameters used can be found in [9].

4.2 Results and Discussion

We used a 24-core IvyBridge server system as baseline for our experiments. As can be seen in Figure 4(a), our accelerator (ACC) outperforms or shows similar performance in 9 out of 17 test cases when compared to the 24-core CPU system. Among 4 applications, PageRank is the best example that can benefit from asynchronous execution and asymmetric convergence [7]. Specifically, our accelerator outperforms the 24-core CPU in 4 cases while having very close execution times in the remaining 2 cases. Additionally, our accelerator shows speedups in the range of 2x to 20x relative to 12 cores. As expected, we observe up to 39% work efficiency which in turn, improves the performance.

The speedup of the LBP application is observed to be between 2.5x and 3x with respect to 24 cores. As shown in [7], LBP-like applications can benefit from asynchronous execution (we have also observed up to 70% work efficiency with our accelerator) thanks to better convergence behavior, but implementing sequential consistency can slow down the execution [10]. For SGD, our accelerator performs better than the 24-core CPU. The reason is that the large number of arithmetic operations per vertex (due to vector product calculation for each edge) can be done more efficiently in custom hardware.

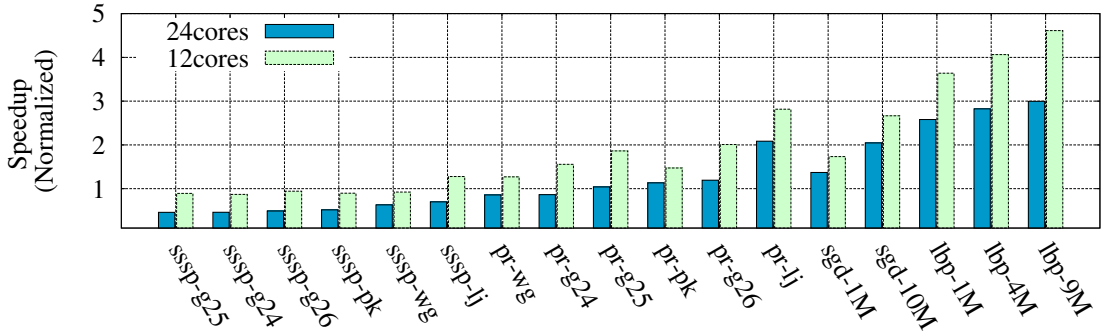
In contrast, SSSP is the only application where we observe worse performance compared to the 24-core CPU, because the CPU implementation uses the delta stepping algorithm, which cannot be modeled by the GAS abstraction. However, we observed that the performance of the accelerator is better than 12 cores for the same kernel.

We observe that power consumed in the accelerator is dominated by the DRAM power as also was shown by previous studies [11]. Note that, DDR3 power is projected to DDR4 power for CPU experiments and we observed that core+uncore power dominates the power consumption for CPU.

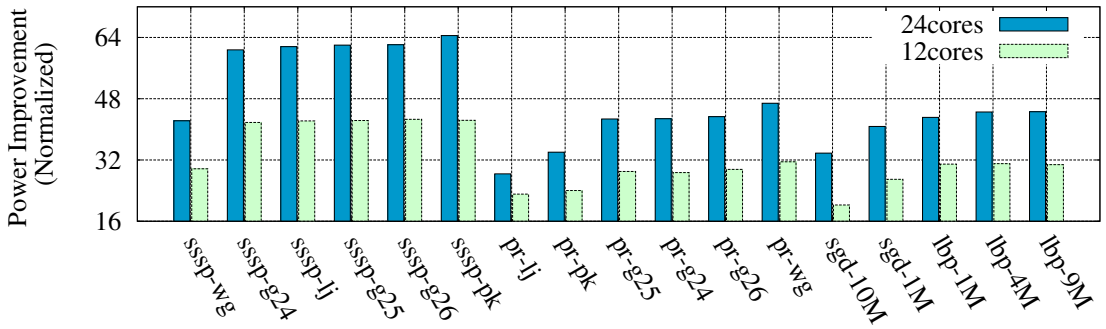
Figure 4(b) shows the power consumption of our accelerators compared to a 24 core CPU. As can be seen from this figure, our accelerator is up to 65x more power efficient. In particular, although SSSP shows worse performance compared to the 24-core system, we observe that it is 65x more power efficient.

Figure 3(a) and 3(b) give the execution time and memory bandwidth as a function of the number of Accelerator Units (AUs). We observe good speedups up to 4 AUs, but beyond this, we see diminishing returns due to the memory bandwidth saturation. When Figure 3(a) and 3(b) are considered, one can see that performance and memory bandwidth utilization follow a similar pattern for irregular graph applications. Note that, PR and SSSP can achieve high memory bandwidth utilization even with 1 or 2 AUs because they require limited amount of computation per edge compared to LBP and SGD.

In conclusion, we have proposed an accelerator architecture targeted at graph analytics applications that follow the well-known Gather-Apply-Scatter abstraction. Due to irregular memory access patterns in these applications, the performance bottleneck is the system memory bandwidth. We have shown that the proposed accelerators can utilize this bandwidth in a much more power efficient way than multi-core CPUs. Furthermore, the proposed template architecture includes work efficiency features targeted at iterative graph applications, which lead to performance improvements of up to 3x. Although we have studied fixed-function accelerators in this paper, it is possible to make the proposed architecture software programmable by replacing the application-specific logic with simple processors in a future work. Another future work is to generalize the proposed architecture beyond the Gather-Apply-Scatter abstraction model.



(a) Execution time normalized to ACC.



(b) Power consumption normalized to ACC.

Fig. 4: Power and performance comparison of ACC and CPU. The y-axis is the speedup/power improvement of the proposed accelerator normalized with respect to accelerator execution.

REFERENCES

[1] "Why cognitive computing solutions, driven by advanced analytics will replace traditional applications," http://www.reltio.com/blog/2015/9/why-cognitive-computing-solutions-driven-by-advanced-analytics-will-replace-traditional-applications.

[2] R. Nambiar and M. Poess, *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference*. Springer, 2016, vol. 9508.

[3] "From knowledge graphs to cognitive computing," https://www.ibm.com/blogs/research/2016/01/from-knowledge-graphs-to-cognitive-computing/.

[4] J. A. Bilmes, "Graphical models and automatic speech recognition," in *Mathematical foundations of speech and language processing*. Springer, 2004, pp. 191–245.

[5] "A case for a situationally adaptive many-core execution model for cognitive computing workloads," http://www.engr.uconn.edu/~omer.khan/pubs/sas-cogarch16.pdf.

[6] R. Mihalcea and P. Tarau, "Textrank: Bringing order into texts." Association for Computational Linguistics, 2004.

[7] 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.

[8] G. Erkan and D. R. Radev, "Lexrank: Graph-based lexical centrality as salience in text summarization," *J. Artif. Int. Res.*, vol. 22, no. 1, pp. 457–479, Dec. 2004.

[9] M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proc. of the 43rd Annual International Symposium on Computer Architecture, to appear*, ser. ISCA '16. IEEE, 2016.

[10] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, S. Burns, and O. Ozturk, "Architectural requirements for energy efficient execution of graph analytics applications," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 676–681.

[11] 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, no. 1, pp. 269–284, Feb. 2014.