Graph Analytics Accelerators for Cognitive Systems

Muhammet Mustafa Ozdal
Bilkent University

Serif Yesil
University of Illinois at Urbana–Champaign

Taemin Kim
Andrey Ayupov
John Greth
Steven Burns
Intel

Ozcan Ozturk
Bilkent University

This article proposes an accelerator architecture specifically optimized for vertex-centric graph applications with irregular memory access patterns, asynchronous execution, and asymmetric convergence. The proposed architecture addresses the limitations of existing CPU and GPU systems while providing a customizable template. Experiments show that the generated accelerators can outperform a high-end CPU system with up to 3 times better performance and 65 times better power efficiency.

Cognitive systems comprise many elements, such as natural language processing, artificial intelligence, machine learning, and data analytics.[1] Due to the increasingly large amounts of data that need to be processed, ongoing efforts are aimed at integrating 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, cognitive systems can use large knowledge bases represented as graphs for reasoning and human interactions.[3]

Many graph algorithms are executed in the inner loops of cognitive systems. Various speech-recognition and language-processing models used in cognitive applications can be represented as graphs[4] (see the "Graph-Based Cognitive Applications" sidebar for more information).

Graph analytics–based cognitive applications differ from traditional computationally intensive applications that have regular access patterns and abundant data and thread-level parallelism. Graph applications are hard to parallelize due to irregular execution patterns and synchronization requirements. In this article, we propose an accelerator architecture that targets graph analytics applications. We implement our architecture as a template to make it easy to model different applications. Architects and designers can plug application-level data structures and functions into this template to generate hardware implementations for a large class of graph analytics applications.

Graph Analytics Applications

Graph analytics applications are among the core algorithms used in cognitive systems. However, efficiently implementing these applications on existing systems is not trivial. There are several reasons for this, including memory access bottlenecks, synchronization problems, and irregular computation and communication patterns. These properties can be exploited to improve performance and

Graph-Based Cognitive Applications

Finite-state decoding graphs are commonly used by WFST-based speech recognition algorithms, wherein the objective is to compute the most likely paths corresponding to the input sequences.[1] Belief propagation on a Bayesian network is another graph analytics application used by different cognitive applications.[2] Cognitive platforms such as unmanned aerial vehicles or self-driving cars run single-source shortest-path (SSSP) algorithms in their inner-loop computations.[3] 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 document tagging based on key phrases and sentence extraction for automatic summarization.[4] TextRank operates on graphs in which vertices represent terms and edges represent the associations inferred from the input texts.

References

1. J.A. Bilmes, "Graphical Models and Automatic Speech Recognition," *Mathematical Foundations of Speech and Language Processing*, Springer, 2004, pp. 191–245.
2. R. Nambiar and M. Poess, eds., *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference*, Springer, 2016, vol. 9508.
3. M. Ahmad, C.J. Michael, and O. Khan, "A Case for a Situationally Adaptive Many-Core Execution Model for Cognitive Computing Workloads," *Proc. 2nd Workshop Cognitive Architectures* (CogArch 16), 2016; www.engr.uconn.edu/~omer.khan/pubs/sas-cogarch16.pdf.
4. R. Mihalcea and P. Tarau, "TextRank: Bringing Order into Texts," *Proc. Conf. Empirical Methods in Natural Language Processing* (EMNLP), 2004, pp. 404–411.

power efficiency by customizing the hardware.

Many graph algorithms are iterative in nature, wherein execution continues until a convergence criteria is met. However, the number of iterations required to converge individual vertices can vary significantly. For instance, Figure 1a plots the percent of vertices not converged throughout the PageRank iterations, where less than 1 percent of vertices require all iterations to converge. Our analysis shows that enabling asymmetric convergence decreases the total number of edges processed by 47 percent on average. Although single-instruction, multiple-data architectures can process multiple vertices simultaneously, they have control-divergence issues when asymmetric convergence is enabled.

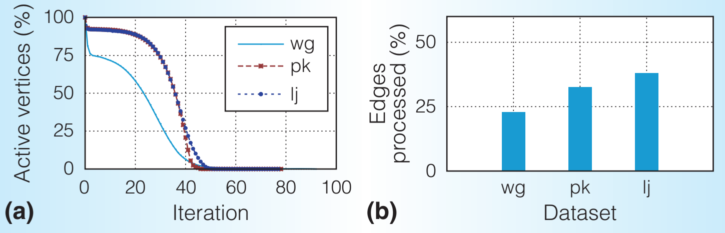

Figure 1. Analysis of the PageRank application on three datasets: web-Google (wg), soc-Pokec (pk), and soc-LiveJournal (lj). (a) Asymmetric convergence of vertices for PageRank. (b) Work efficiency when asymmetric convergence and asynchronous execution features are enabled.

Asynchronous execution, which lets neighboring vertices access the most recent data, can improve work efficiency even further.[5] Figure 1b shows the relative number of edges processed for PageRank when support for both asynchronous execution and asymmetric convergence is enabled. These features reduce the number of edges processed by 66 percent on average. However, programmers who want to utilize the work efficiency of asynchronous execution should handle synchronization and enforce sequential consistency in software to prevent data race conditions. Fine-grained locking mechanisms slow down the execution of both CPUs and throughput architectures.

One main bottleneck in graph applications is memory access due to low computation-to-communication ratios, low spatial and temporal locality, and hard-to-predict memory accesses.

Some real-world graphs, such as social networks, follow power-law distribution, wherein a small number of vertices have

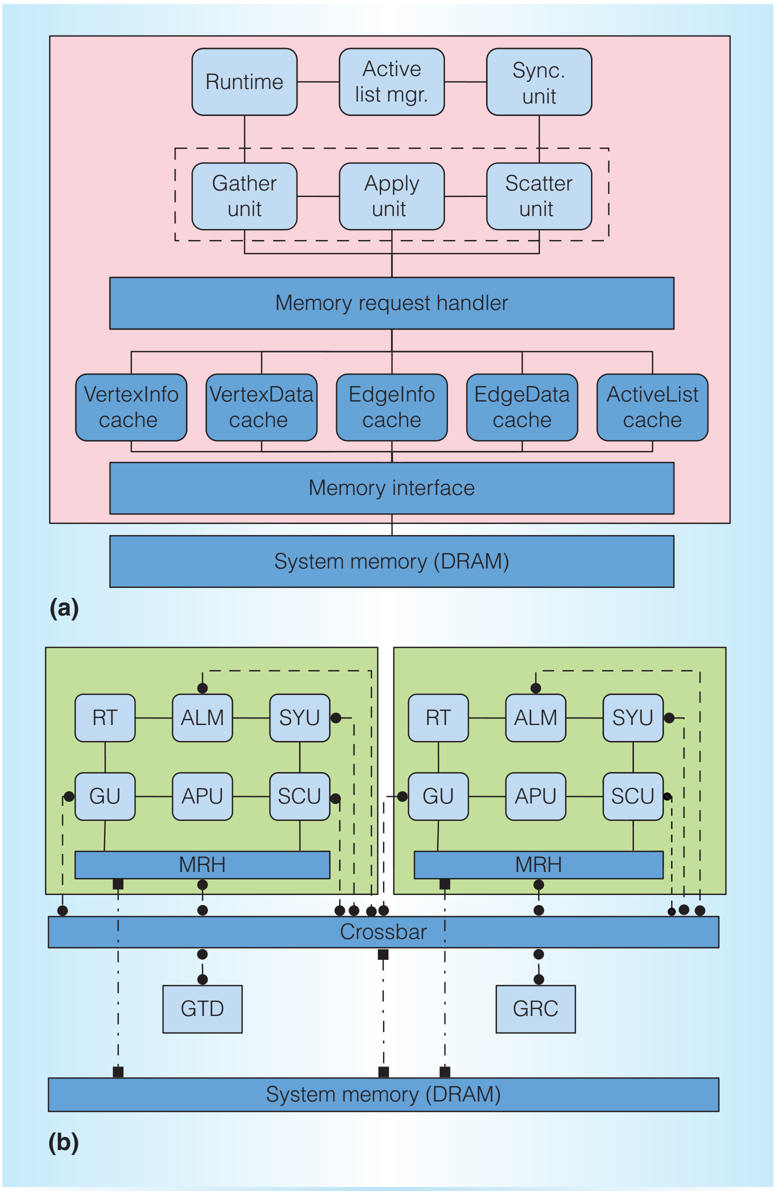

Figure 2. Accelerator block diagram. For clarity, some of the connections between the blocks are not shown. (a) Single accelerator unit. (b) Multiple accelerator units.

much higher degrees when compared to the rest. In such an environment, static partitioning of vertices to different threads greatly suffers from load imbalances. An efficient implementation must distribute its workload carefully, and more importantly, should be able to handle high-degree vertices efficiently.

Proposed Architecture Template

Several graph frameworks have been proposed in recent years. The common objective is to hide the complexities of parallel and distributed software development by providing a 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, which comprises gather-apply-scatter functions as in GraphLab.[5] In this model, users must define basic data structures corresponding to each vertex and 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 and 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, whereas the common architecture template is more than 30,000 lines of SystemC code and is not visible to the user.

The proposed accelerator is loosely coupled with the host processor and is connected to the system DRAM. We assume 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 2a shows the proposed high-level architecture for a single accelerator unit (AU). The architecture has the following features:

- 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.
- 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.
- Synchronization between concurrently processed vertices and edges is done in the sync unit (SYU) module, which is designed for graph processing. This module ensures sequential consistency with negligible performance overhead. Furthermore, it works in a distributed fashion without a centralized bottleneck.
- The active list manager (ALM) module maintains the set of active (not yet converged) vertices. This module enables simultaneous high-throughput reads and writes to and from the distributed active list (AL) data structure without the need for expensive locking mechanisms.
- The memory subsystem is optimized for sparse graph data structures.

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

Computational Units

The main computational units in our accelerator are the gather unit (GU), apply unit (APU), and scatter unit (SCU), which are shown in Figure 2a. 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, in which many vertices and edges are processed concurrently, and partial vertex and edge states are stored locally.

The limited local storage available in the GU is shared among all concurrently processed vertices. In our proposed GU microarchitecture, a credit-based mechanism assigns 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 made dynamically on the basis of the vertex degrees and priorities.

The APU module performs computation for each vertex using the data computed by the 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.

The SCU implements the scatter program for each vertex *v*, in which the application-specific scatter functions determine how to distribute the updated data of *v* to its neighbors. Similar to the 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 neighboring vertex *u* of vertex *v*, the application-specific function also determines whether *v* should activate *u* (that is, schedule *u* for future execution).

Enabling Sequential Consistency

The SYU is the critical module that allows race-free and sequentially consistent execution of all vertices in the proposed architecture. The 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,[5] 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.

The SYU microarchitecture comprises the following basic operations.

Maintain vertex states. Once a new vertex is received from runtime, it is assigned a unique rank and stored in a table that 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. The execution state of v is also updated when a gather-done or scatter-done message is received.

Maintain RAW ordering. Consider an edge $e : u \rightarrow v$, in which rank(u) < rank(v)—that is, the execution of u should (logically) happen before that of 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 the GU go through the SYU to ensure this ordering.

Maintain WAR ordering. Consider edge $u \rightarrow v$. There is a potential WAR dependency between u and v if rank(u) > rank(v). To maintain WAR ordering, the SCU sends a message to the 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 the SCU corresponding to edge $u \rightarrow v$. This implies that vertex v should be added to the 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. The SYU filters out such unnecessary activations before passing the activation requests to the ALM.

Managing Active Vertices

The 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. Because the AL could contain all vertices in the input graph, it must be stored in the system memory. As we explained earlier, the application-specific convergence condition is checked in the SCU to determine which vertices to schedule for future execution, whereas the unnecessary activations are filtered out in the SYU. The ALM is responsible for the following tasks: extracting vertices from the AL and sending them to runtime for execution, and receiving new activation requests from the SYU and adding them to the AL while avoiding duplications.

For storage and data-access efficiency, the AL comprises two data structures: a bit vector in which each bit corresponds to the presence or absence of a vertex in the AL, and a queue of bit vector indices in which each index corresponds to a 256-bit segment of the bit vector.

To extract new vertices for execution, the 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 have set bits in the bit vector to runtime for execution.

When the ALM receives an activation request for vertex v, it first checks whether the bit corresponding to v is locally stored in the ALM. If so, it simply sets that bit locally. Otherwise, it sends the request to the AL memory unit. Special care must be taken to handle in-flight bit vectors and vertex indices. Specifically, when a vertex index is sent to runtime, it also must be registered with the SYU, 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 could fail to detect that the vertex is already being executed. Similarly, the in-flight bit vectors between ALM and AL memory must be handled with care to avoid adding duplicate vertices to AL.

Runtime

The runtime (RT) module is in charge of monitoring available resources in the AU and scheduling new vertex executions. It reads new vertices from the ALM and sends them to the SYU when it detects that there are available resources. It is also responsible for detecting the termination condition and sending out a completion signal when there are no in-flight or executing vertices and the

AL is empty. Runtime is a simple module comprising two counters to track the number of vertices in the gather and scatter stages.

Specialized Memory Subsystem

Different data structures must be accessed when a vertex program is executed. In this article, we assume that the popular compressed sparse row 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 Edge Info. The offsets to this array are stored in a separate array denoted as Vertex Info. In addition, application-specific data structures can be defined per vertex and edge, which are denoted as Vertex Data and Edge Data. As we explained previously, the AL must also be stored in main memory.

In the proposed architecture, we define a custom cache corresponding to each graph object type as shown in Figure 2a. The access patterns for different object types can vary significantly. For example, Edge Info accesses tend to have good spatial locality because of contiguous storage of indices. On the other hand, Vertex Data and Edge Data 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 template-based architecture, and they can be determined based on the specific application requirements.

Multiple Accelerator Units

We can further improve throughput by replicating the AUs as shown in Figure 2b. In this article, we focus on fine-grained parallelism by tightly integrating a small number of AUs, and statically assigning vertices and edges to AUs on the basis of their indices. The memory subsystem is also partitioned according to this assignment in a multibank fashion.

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

The first is the global rank counter (GRC) module. As we've described, sequential consistency is implemented by assigning monotonically increasing unique ranks to vertices. When multiple AUs are involved, monotonicity is achieved by a 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. The GRC is connected to each AU's SYU.

The second module is the global termination detector (GTD). Each AU's runtime is responsible for detecting the termination condition for that AU. When multiple AUs are involved, the GTD collects the termination signals from individual RTs and determines the termination condition of the whole system. The GTD is responsible for notifying the host processor that the computation is finished.

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

Empirical Study

We selected four graph analytics applications that are used as part of cognitive systems.

- PageRank (PR) is a ranking algorithm that is used not only to rank web pages, but also to summarize text, as in TextRank and LexRank.[6]
- Stochastic gradient descent (SGD) is an iterative machine-learning algorithm that is used in personalized recommendations. This is especially important because many web services depend on personalized recommendations to improve user experience.
- Single-source shortest path (SSSP) is a kernel used in unmanned aerial vehicles, which are among future cognitive systems, and also in network analytics as a kernel for betweenness-centrality calculations.
- Loopy belief propagation (LBP) is an important kernel used by cognitive systems in the context of image processing and other applications.

For each benchmark, we tried to use the most efficient parallel implementation, either by obtaining from available benchmark suites or manually optimizing. Our preliminary

Table 1. Datasets used in our experiments.[7]

Application	Dataset	No. of vertices	No. of edges
PageRank and single-source shortest-path (SSSP) (directed)	web-Google (wg)	916,000	5.1 million
	soc-Pokec (pk)	1.6 million	30 million
	soc-LiveJournal (lj)	4.8 million	69 million
	g24 (synthetic)	16.8 million	268 million
	g25 (synthetic)	33.5 million	536 million
	g26 (synthetic)	67 million	1 billion
Loopy belief propagation (undirected)	1,000 × 1,000 pixels	1 million	2 million
	2,000 × 2,000 pixels	4 million	8 million
	3,000 × 3,000 pixels	9 million	18 million
Stochastic gradient descent (undirected)	1 million ratings	9,700	1 million
	10 million ratings	80,000	10 million

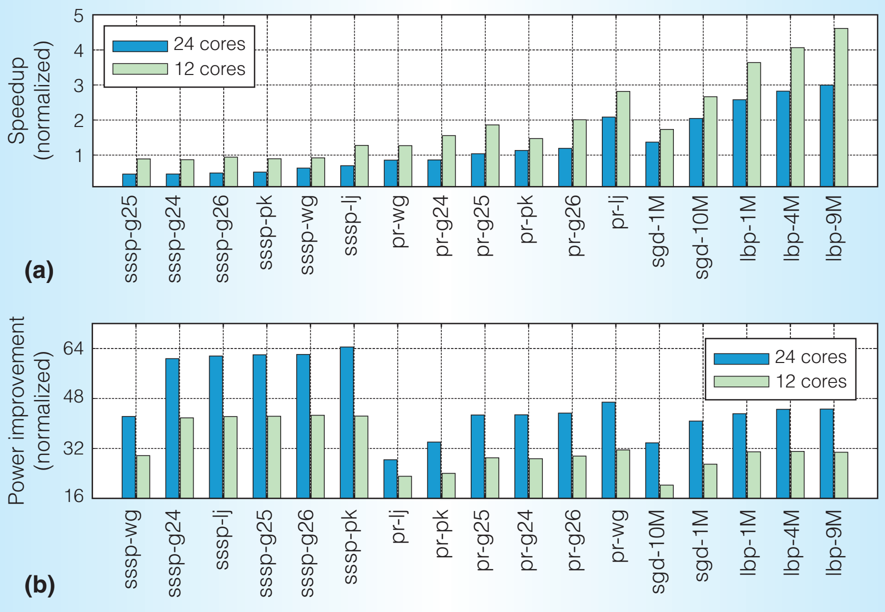

Figure 3. Power and performance comparison of the accelerator (ACC) and CPU. The *y*-axis is the speedup and power improvement of the proposed accelerator normalized with respect to accelerator execution. (a) Execution time. (b) Power consumption.

work offers further details of applications and selected benchmark implementations.[7]

Experiments

We tested each application with various datasets from well-known graph databases.[7] The number of vertices and edges in the selected input graphs for PageRank and SSSP applications ranged from 916,000 vertices and 5.1 million edges to 67 million vertices and 1 billion edges. For LBP, we used three images, with (1,000 × 1,000), (2,000 × 2,000), and (3,000 × 3,000) pixels. For SGD, we selected two movie datasets with 1 million and 10 million ratings. Table 1 shows the details of the selected datasets.

To calculate the native system's energy and power consumption, we used the running average power limit. For our accelerator implementations, we used 22-nm libraries for standard cells and metal layers, Cacti for caches, and DramSim2 for memory. We used a commercial high-level synthesis 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 was four. However, other microarchitectural parameters (such as cache sizes and the number of vertices and edges processed concurrently) have been tuned individually per application. Our preliminary work offers further details of our experimental setup, including the parameters used.[7]

Results and Discussion

We used a 24-core IvyBridge server system as the baseline for our experiments. As Figure 3a shows, our accelerator (ACC) outperforms or shows similar performance in 9 out of 17 test cases when compared to the 24-core CPU system. Among four applications, PageRank is the best example that can benefit from asynchronous execution and asymmetric convergence.[5] Specifically, our accelerator outperforms the 24-core CPU in four cases, while having very close execution times in the remaining two cases. Additionally, our accelerator shows speedups in the range of 2 to 4.6 times relative to 12 cores. As expected, we have observed up to 39 percent work efficiency, which in turn improves the performance.

The speedup of the LBP application is between 2.5 and 3 times with respect to 24 cores. LBP-like applications can benefit from asynchronous execution (we have observed up to 70 percent work efficiency with our accelerator) thanks to better convergence behavior,[5] but implementing sequential consistency can slow down the execution on a CPU.[8] For SGD, our accelerator performs better than the 24-core CPU because 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 that performed worse than the 24-core CPU, because the CPU implementation uses the delta stepping algorithm, which cannot be modeled by the gather-apply-scatter abstraction.

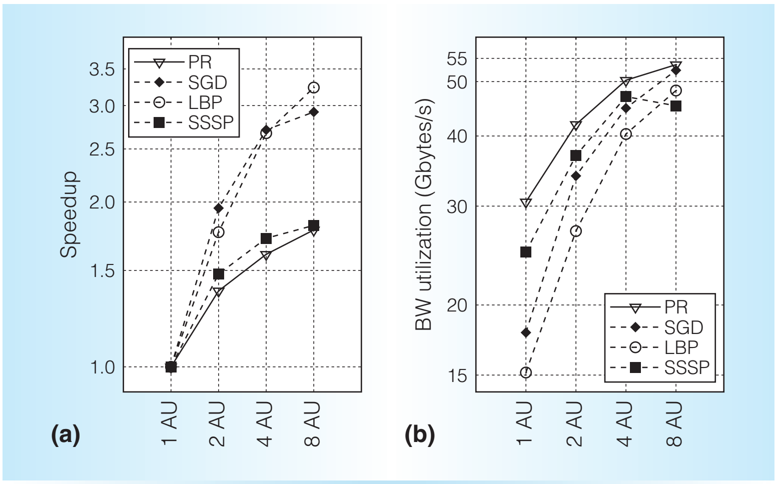

Figure 4. Accelerator scalability in terms of execution time and memory bandwidth utilization (normalized with respect to one accelerator). (a) Execution time. (b) Bandwidth.

We observed that power consumed in the accelerator is dominated by the DRAM power, as previous studies have also shown.[9] DDR3 power is projected to DDR4 power for CPU experiments, and core + uncore power dominates the CPU's power consumption.

Figure 3b shows the power consumption of our accelerators compared to a 24-core CPU. Our accelerator is up to 65 times more power efficient. In particular, although SSSP performed worse than the 24-core system, it was 65 times more power efficient.

Figures 4a and 4b show the execution time and memory bandwidth, respectively, as a function of the number of AUs. We observed good speedups for up to four AUs, but beyond this, we saw diminishing returns due to the memory bandwidth saturation. Figure 4 shows that performance and memory bandwidth utilization follow a similar pattern for irregular graph applications. PageRank and SSSP can achieve high memory bandwidth utilization even with one or two AUs, because they require limited computation per edge compared to LBP and SGD.

Our accelerator architecture targets 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 use this bandwidth in a much more-power efficient way than multicore CPUs. Furthermore, the proposed template architecture includes work efficiency features targeted at iterative graph applications, which lead to performance improvements of up to 3 times. Although we have studied fixed-function accelerators in this article, future work could make the proposed architecture software programmable by replacing the application-specific logic with simple processors. Another aim of future work is to generalize the proposed architecture beyond the gather-apply-scatter abstraction model. MICRO

Acknowledgments

A preliminary version of this article was published in the *Proceedings of the 2016 ACM/IEEE International Symposium on Computer Architecture* (ISCA).[7] This project has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement no 704476.

References

1. J. Hurwitz, "Why Cognitive Computing Solutions, Driven by Advanced Analytics Will Replace Traditional Applications," *Reltio*, blog, 21 Sept. 2015; www.reltio.com/blog/2015/9/why-cognitive-computing-solutions-driven-by-advanced-analytics-will-replace-traditional-applications.
2. R. Nambiar and M. Poess, eds., *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference*, Springer, 2016, vol. 9508.
3. L. Sbodio, "From Knowledge Graphs to Cognitive Computing," *IBM Research Blog*, 18 Jan. 2016; www.ibm.com/blogs/research/2016/01/from-knowledge-graphs-to-cognitive-computing.
4. J.A. Bilmes, "Graphical Models and Automatic Speech Recognition," *Mathematical Foundations of Speech and Language Processing*, Springer, 2004, pp. 191–245.
5. Y. Low et al., "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *Proc. VLDB Endowment*, vol. 5, no. 8, 2012, pp. 716–727.
6. G. Erkan and D.R. Radev, "LexRank: Graph-Based Lexical Centrality as Salience in Text Summarization," *J. Artificial Intelligence Research*, vol. 22, no. 1, 2004, pp. 457–479.
7. M. Ozdal et al., "Energy Efficient Architecture for Graph Analytics Accelerators," *Proc. 43rd Ann. Int'l Symp. Computer Architecture*, 2016, pp. 166–177.
8. M.M. Ozdal et al., "Architectural Requirements for Energy Efficient Execution of Graph Analytics Applications," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design*, 2015, pp. 676–681.
9. T. Chen et al., "Diannao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," *SIGARCH Computer Architecture News*, vol. 42, no. 1, 2014, pp. 269–284.

Muhammet Mustafa Ozdal is an assistant professor in the Department of Computer Engineering at Bilkent University. His research interests include high-performance computing, heterogeneous architectures, and hardware accelerators. Mustafa Ozdal received a PhD in computer science from the University of Illinois at Urbana–Champaign. Contact him at mustafa.ozdal@cs.bilkent.edu.tr.

Serif Yesil is a PhD student in the Computer Science Department at the University of Illinois at Urbana–Champaign. His research interests include computer architecture, heterogeneous architectures, and parallel computing. Yesil received an MS in computer engineering from Bilkent University. Contact him at syesil2@illinois.edu.

Taemin Kim is a research scientist in Strategic CAD Labs at Intel. His research interests include accelerator architecture synthesis from high-level specifications. Kim received a PhD in computer engineering from North Carolina State University. He is a member of IEEE and ACM. Contact him at taemin.kim@intel.com.

Andrey Ayupov is a research scientist in Strategic CAD Labs at Intel. His research interests include hardware agile design, high-level synthesis, and accelerator design. Ayupov received a PhD in computer engineering from the Moscow Institute of Physics and Technology. Contact him at andrey.ayupov@intel.com.

John Greth is a logic designer in the Data Center Group at Intel. His research interests include quantitative economics, macroeconomics, and organizational development. Greth received an MS in computer and electrical engineering from Carnegie Mellon University. Contact him at john.greth@intel.com.

Steven Burns is a senior principal engineer in Strategic CAD Labs at Intel. His interests include large-scale optimization, effective utilization of advanced process technologies, and software-centric flows for designing accelerator hardware. Burns received a PhD in computer science from Caltech. Contact him at steven.m.burns@intel.com.

Ozcan Ozturk is an associate professor in the Department of Computer Engineering at Bilkent University. His research interests include accelerators, many-core architectures, parallel computing, and computer architecture. Ozturk received a PhD in computer science and engineering from Pennsylvania State University. Contact him at ozturk@cs.bilkent.edu.tr.