GAS: A Heterogeneous Memory Architecture for Graph Processing

Minxuan Zhou, Mohsen Imani, Saransh Gupta, and Tajana Rosing
CSE Department, UC San Diego, La Jolla, CA 92093, USA
{miz087, moimani, sgupta, tajana}@ucsd.edu

ABSTRACT

Graph processing has become important for various applications in today's big data era. However, most graph processing applications suffer from large memory overhead due to random memory accesses. Such random memory access pattern provides little temporal and spatial locality which cannot be accelerated by the conventional hierarchical memory system. In this work, we propose GAS, a heterogeneous memory architecture, to accelerate graph applications implemented in message-based vertex program model, which is widely used in various graph processing systems. GAS utilizes the specialized content-addressable memory (CAM) to store random data, and determine exact access patterns by a series of associative search. Thus, GAS not only removes the inefficiency of random accesses but also reduces the memory access latency by accurate prefetching. We test the efficiency of GAS with three important graph processing kernels on five well-known graphs. Our experimental results show that GAS can significantly reduce cache miss rate and improve the bandwidth utilization as compared to a conventional system with a state-of-the-art graph-specific prefetching mechanism. These enhancements result in 34% and 27% reduction in energy consumption and execution time, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Hardware** → *Emerging technologies*;

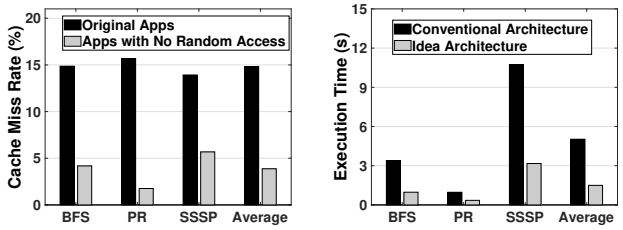
ACM Reference Format:
Minxuan Zhou, Mohsen Imani, Saransh Gupta, and Tajana Rosing. 2018. GAS: A Heterogeneous Memory Architecture for Graph Processing. In *ISLPED '18: ISLPED '18: International Symposium on Low Power Electronics and Design, July 23–25, 2018, Seattle, WA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3218603.3218631

1 INTRODUCTION

Graph has become one of the most important data structures in today's big-data era. Graph frameworks have been proposed to reduce the difficulty of implementing different graph processing applications on various systems [1, 2]. However, previous work [3–5] found that conventional architecture is inefficient in processing graph-based applications. Such inefficiency mainly comes from the inconsistency between the conventional memory hierarchy and the random memory access pattern existing in graph processing applications.

ISLPED '18, July 23–25, 2018, Seattle, WA, USA

ACM ISBN 978-1-4503-5704-3/18/07…$15.00
https://doi.org/10.1145/3218603.3218631



(a) Cache miss rate (b) Execution Time

Figure 1: Effect of random memory accesses.

Algorithm 1 shows the message-based vertex program model, which is widely used in several popular graph processing systems [2, 6]. The property (*vProp*) and temporary property (*vTProp*) are application-specific values for each vertex. Messages represent necessary information for updating the *vProp*s and *vTProp*s. The messages are usually sent to each vertex from its in-coming neighbors. In each iteration, an *ActiveList* consists of vertices to be processed for producing out-going messages based on an application-specific *Process* function. Each vertex updates its *vProp* based on in-coming messages and two other functions - *Reduce*, and *Apply*. Every vertex with an updated *vProp* is added to the active list in the next iteration. The application is completed when there is no active vertex. In this work, we investigate three important graph kernels implemented in the framework, including bread-first-search (BFS), page-rank (PR) and single-source-shortest-path (SSSP). Table 1 shows the detailed implementations of these three applications.

In the vertex program model, accessing messages of each destination vertex during the *Gather* phase is random because messages are stored based on the source vertex of each edge. Such random memory access pattern experiences poor spatial locality which causes a large number of cache misses. High cache miss rate makes the processor frequently communicate with the slow off-chip memory. Figure 1a shows the average L1 cache miss rate when testing three graph processing applications on five graphs including three real-world graphs and two synthetic graphs. We compare the original cache miss rate with the case when there is no random access (by eliminating cache accesses during message gathering). The result shows that removing random accesses can decrease cache miss rate by 11.0%. To further illustrate the impact of cache miss rate on the performance, we compare the performance of conventional architecture and ideal architecture, which assumes that all random memory accesses can be served by L1 cache (without influencing cache behaviours of other accesses). Figure 1b shows that ideal architecture can reduce the execution time by 68.7% on average. Both of these results indicate that graph processing can be significantly improved by processing the message gathering more efficiently.

Table 1: Framework-based implementation of different applications

Application	*vProp*	*Process*(u,e)	*Reduce*(u,e)	*Apply*(u)
Breadth-First-Search	Level	$e.msg = u.vProp + 1$	$u.vTProp = min(e.msg, u.vTProp)$	$u.vProp = min(u.vProp, u.vTProp)$
Page Rank	Page rank score	$e.msg = u.vProp * u.out\_degree\_factor$	$u.vTProp = u.vTProp + e.msg$	$u.vProp = a * v.vTProp + base$
Single-Source-Shortest-Path	Distance	$e.msg = u.vProp + e.weight$	$u.vTProp = min(e.msg, u.vTProp)$	$u.vProp = min(u.vProp, u.vTProp)$

Algorithm 1: Framework of vertex program model

```
1: while ActiveList is not empty do
2:     for v : ActiveList do
3:         for e : OutEdges(v) do
4:             e.msg = Process(v.vProp, e)                ▷ Scatter
5:         end for
6:     end for
7:     for v : AllVertices do
8:         for e : InEdges(v) do
9:             v.vTProp = Reduce(v.vTProp, e.msg)         ▷ Gather
10:        end for
11:    end for
12:    for v : AllVertices do
13:        v.vProp = Apply(v.vProp, v.vTProp)             ▷ Apply
14:        UpdateActiveList(v.vProp != v.vTProp)
15:    end for
16: end while
```

In this work, we propose GAS, a heterogeneous memory architecture which utilizes various memory techniques to efficiently handle messages in the vertex program model. Other than components in conventional systems (cache and main memory), GAS consists of three main components: an on-chip buffer, a meta-data table, and an addressable GAS memory. GAS handles a special memory region managed by GAS memory to store all generated messages. At the same time, the meta-data table is set up to store the basic information of messages including newly allocated memory addresses. The meta-data table is stored in CAM which supports associative search, which is exploited to retrieve information of in-coming messages of each vertex. A prefetching mechanism based on the search results is proposed to load the corresponding data to the fast on-chip buffer before the data is requested. We also design the software interface to allow programmers easily implement graph applications accelerated by GAS. We test GAS over 3 important graph processing kernels on five real-world graphs, and the results show GAS can reduce the number of cache misses by 44% and improve the bandwidth utilization by 2.7× compared to a conventional system with a state-of-the-art prefetching mechanism. Such improvement results in 34% and 27% reduction in energy consumption and execution time.

2 RELATED WORK

Several prior works tried to accelerate data-intensive applications by enabling processing in-memory [7–11]. The in-memory computation often enables on non-volatile memory with high density and zero leakage power [12, 13]. In framework-based graph processing applications, the computation has been accelerated using 3D DRAM [14], memrisitor crossbar [15], and ASIC design[4]. All these customized graph accelerators require significant changes to the existing computer system to adopt them. On the contrary, GAS is a new memory architecture that manages graph data in heterogeneous memory devices. The memory devices can be designed and adopted based on existing technologies. Computer systems can deploy GAS with acceptable overhead. In addition, GAS can also be deployed in accelerators which utilize message-passing based model to further improve their performance.

Several prefetching mechanisms have been proposed to accelerate applications with irregular access patterns like graph processing applications [16, 17]. However, such prefetching mechanisms are not stable to recognize the target memory access patterns which usually depend on a lot of factors like algorithm, compiler, operating system and architecture. Unlike these prefetching mechanisms, GAS do not predict the access pattern. Instead, GAS utilizes the associative search supported by CAM to accurately prefetch data required by applications. Users can utilize the simple interface to accelerate their applications without considering platform-specific factors.

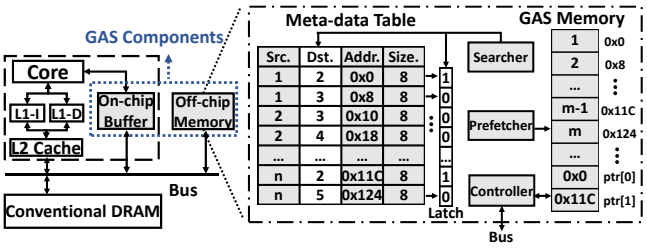


Figure 2: Overall architecture of GAS.

3 GAS DESIGN

GAS is an extension of the conventional memory system, which is designed to accelerate vertex program graph applications by optimizing the memory accesses to messages. In this section, we introduce the detailed design of GAS.

3.1 Overall Architecture

Figure 2 shows the overall architecture of GAS integrated in a conventional system with two-level caches and an off-chip DRAM. GAS consists of two hardware components: an off-chip memory and an on-chip buffer. We store all messages in GAS memory without going through the conventional memory hierarchy considering the fact that accessing messages is inefficient. The processor communicates with GAS through the on-chip buffer and a corresponding controller to handle all memory accesses to data stored in the off-chip addressable GAS memory. On-chip buffer is used to cache the data transferred between the processor and GAS memory using the system bus.

Off-chip GAS memory consists of five main components: addressable GAS memory, controller, prefetcher, searcher and meta-data table. GAS memory can be accessed in the same address space as the DRAM memory. Programmer calls specific interface to allocate the memory space for messages in this memory region. The controller issues memory commands to the GAS memory. The meta-data table is stored in CAM blocks supporting associative search. Each entry of the meta-data table contains the basic message information. The order of messages stored in meta-data table is determined in *Scatter* phase, which is based on the order of out-going edges of active vertices. An associative search can be issued for each vertex in the meta-data table to find out all in-coming messages of the vertex. The result of each search contains the address information of future memory accesses, which can be exploited to improve the performance by prefetching. Associative search and prefetching are

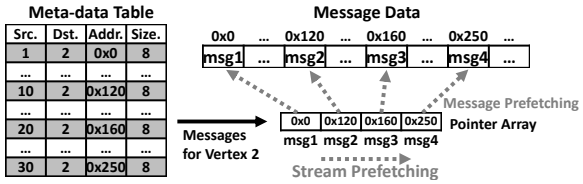


Figure 3: Pointer and message prefetching

implemented in the searcher and the prefetcher respectively. The searcher issues search commands to the meta-data table and handles the search results. The prefetcher notifies the controller to load extra data to the on-chip buffer. The detailed of the prefetching mechanism are introduced in Section 3.3.

3.2 Memory Management

GAS manages a special memory region in the same address space as the DRAM memory. Figure 2 shows an example of how GAS manages its memory space during the *Scatter* phase. For each active vertex, the program checks all of its out-going edges and calculates a message for each edge based on the application-specific *Process* function. For each message, the source vertex, the destination vertex, and the message data are sent to GAS. GAS allocates the data to a memory location inside the GAS memory. Then, the source vertex, the destination vertex, the allocated address are used to set up an entry in the meta-data table.

As mentioned earlier, the meta-data table is a CAM which supports associative search. Addresses of the generated messages sent to a specific vertex (2 in the figure) can be fetched by searching the meta-data table. The query will activate all entries in the table, whose destination vertex column is the target vertex. The message addresses of all activated entries are stored in the GAS memory as a pointer array. Then, the pointer array is returned to the application to access all messages sent to the vertex. Thus, accessing messages of a target vertex in GAS requires two stages. At first, GAS stores a pointer array of in-coming messages based on associative search results, and returns the address of the array to the program. In the second stage, the program checks each message required in the *Gather* phase by traversing the pointer array.

3.3 Prefetching

We propose a hardware prefetching mechanism, which fully utilizes the on-chip buffer and the off-chip GAS memory bandwidth, to accelerate memory accesses in the pointer array and message data. After searching the meta-data table for a vertex, we get a pointer array of all in-coming messages sent to this vertex. With the pointer array generated by an associative search, the access pattern happens during *Gather* is known because a vertex gathers messages from all its in-coming edges. Accessing a message requires two memory accesses: the access to the message pointer, and the access to the message data denoted by the pointer. Since the access pattern of the pointer array is in-order, the next few addresses are requested in the near future. Furthermore, the content of the address denoted by each pointer is accessed by the application for loading the actual message data. Based on these two observations, GAS prefetches not only the next few pointers (stream prefetching), but also the data stored in these addresses (message prefetching). The application can access both the next message pointer and the corresponding message data from the on-chip buffer. Figure 3 shows the example of prefetching.

In GAS, prefetching operations are triggered by a miss in the on-chip buffer. If a miss happens, the buffer controller loads data from the off-chip GAS memory. The GAS memory controller loads the data based on the address of the miss. If the miss happens in the pointer array, GAS issues both pointer and message prefetches to the on-chip buffer. The prefetcher determines how many pointers and corresponding messages should be loaded to the on-chip buffer during one prefetching operation based on the message size (known in the message entry) and the memory bandwidth. If the cache miss happens outside the pointer array, which means a miss on message data, the prefetcher loads the next a few cachelines of that address because such miss may be caused by a large message size. GAS does not prefetch the pointer array since it cannot predict the size of message data without knowing the current message information.

4 IMPLEMENTATION

In this section, we introduce the memory technologies we utilize to implement different hardware components and the software interface provided for programmers to use GAS architecture.

4.1 Memory Implementation

On-chip buffer: Data in the on-chip buffer should be accessed much faster than that in the GAS memory. Since a specific message is only accessed once in the *Gather* phase and the access pattern of messages is not streaming, we implement the on-chip buffer as a fully-associative cache with LRU replacement policy. Because the buffer data (including the pointer array and message) is no longer useful once it's accessed and the off-chip memory bandwidth is limited, the setting of on-chip buffer is similar to a conventional L1 cache. Thus, SRAM technology is used to implement such small and fast on-chip buffer for providing the optimized performance and the tag is stored in the same SRAM chip. The on-chip buffer is handled by a specialized cache controller. Considering both the pointer array and message data will not be updated by the application, there is no coherence issues between the on-chip buffer and the off-chip memory. Removing the coherence management even makes the on-chip buffer more efficient than a conventional cache.

GAS memory: We utilize memristor-based crossbar memory to implement the GAS memory. The organization of GAS memory is similar to a conventional DRAM DIMM where each channel consists of multiple ranks and each rank consists of multiple banks. We assume a single channel in this work, and a memory controller is used to issue load and store commands. We choose memristors because resistive crossbar memory provides better energy-efficiency and density than conventional DRAM/SRAM technologies while still providing acceptable reading performance [18, 19]. Furthermore, accessing messages in graph processing applications does not trigger intensive write operations, so the problems caused by limited endurance and inefficient writes are minimized.

Meta-data table: In this work, we use a resistive CAM because of its high-density and energy efficiency. The architecture of GAS utilizes crossbar resistive CAM [20–22] where each CAM row stores one meta-data entry. Once we issue a search, the CAM sense amplifier detects rows which exactly match with a query signal. In each CAM block, the latch stores the result of each search and it is used to activate the corresponding rows of the same memory block. Next, the memory sense amplifier sequentially reads the message address in each activated row and store them in the write register. Finally, the result is sent to GAS memory.

Algorithm 2: Example of SSSP

```
1: while ActiveList is not empty do
2:     for v : ActiveList do
3:         for e : OutEdges(v) do
4:             alloc_msg(v, e.dst, v.vProp + e.len)                ▷ Scatter
5:         end for
6:     end for
7:     for v : AllVertices do
8:         msgs = load_msgs(v)
9:         for msg : msgs do
10:            v.vTProp = min(v.vTProp, msg− > data)               ▷ Gather
11:        end for
12:    end for
13:    for v : AllVertices do
14:        v.vProp = min(v.vProp, v.vTProp)                        ▷ Apply
15:        UpdateActiveList(v.vProp! = v.vTProp)
16:    end for
17: end while
```

Since the performance of CAM decreases exponentially as the number of CAM rows increases [22], a single CAM block is not enough to store all entries in the meta-data table. Thus, we utilize multiple CAM blocks, each of which has the independent search logic circuit, to store big meta-data tables. For each search, GAS issues commands to all CAM blocks and the result of each CAM block is stored locally. The program accesses CAM blocks one by one to collects all information.

4.2 Software Interface

We propose two functions for programmers to easily utilize GAS to accelerate graph processing. *alloc_msg* allocates a specific message to GAS memory and set up a meta-data entry. Programmer calls *load_msg* to access messages sent to each vertex. Algorithm 2 shows an example of SSSP using GAS.

alloc_msg(src, dst, msg): During the *Scatter* phase, each active vertex calculates a message for each of its out-going edges. For each message, the program calls *alloc_msg* to allocate a memory space in GAS and add the corresponding entry in the meta-data table. This function also requires the programmer to pass the source vertex, the destination vertex, and the data of message to add an entry in the meta-data table. The store instructions of messages are very similar to normal memory accesses except they are handled by the memory hierarchy of GAS. The memory location allocated to messages are continuous which makes storing messages to GAS efficiently using on-chip buffer and direct-memory-access.

load_msgs(v): Each call of *load_msgs* issues a search command on the meta-data table. GAS stores the search result as a pointer array and return it to the program. During the *Gather* phase, each vertex checks all received messages by calling *load_msgs*. Only the target vertex *v* is required for calling *load_msgs*. The program utilizes the pointer array to access all messages sent to each vertex.

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

Simulation Infrastructure: In this work, we use Sniper [23], an accurate and high-speed x86 simulator, for performance simulations. We implement the handler of all memory accesses to message in the Pin-based [24] front-end of Sniper and simulate the behavior of GAS when Sniper handles those accesses. We use McPAT [25] for energy simulation. The method of integrating McPAT with Sniper and simulating energy consumption is the same as that used in the previous work [26]. The timing and power simulations for on-chip buffer are done with CACTI [27].

Table 2: System configuration

	Configuration
Technology	45nm, 1.2V 2.66GHz
Cores	2.66GHz Intel Nehalem-like
L1-D Cache	32 KB, private, 5 cycles, 64B blocks
L1-I Cache	32 KB, private, 5 cycles, 64B blocks
L2 Cache	256 KB, private, 11 cycles, 64B blocks
Off-chip DRAM	30GB/s, latency = 122 cycles
GAS (GAS Memory (memristor))	1GHz bus frequency, 30GB/s read-write : 2-31 cycles
GAS (On-chip Buffer)	32KB, fully-associativity, 32B blocks
GAS (ReRAM CAM)	18B * 4096 per block read-write-search : 2-31-44 cycles

Table 3: Graph workload summary

Graph	#Vertices	#Edges	Description
Wikipedia	3.56M	101M	English part of Wikipedia [30, 31]
Live Journal	5.4M	79M	LiveJournal [30, 31]
Twitter	41M	91M	Twitter [30, 31]
Uniform	2.1M	33M	Uniform random graph (degree 16) [32]
G500	2.1M	32M	Kronecker graph (Graph500 specifications [32, 33])

Table 4: Energy consumption and execution time normalized to the IMP system.

Applications	GAS w/o Prefetching		GAS		Ideal Architecture	
	Energy consum.	*Execution time*	*Energy consum.*	*Execution time*	*Energy consum.*	*Execution time*
BFS	0.67	1.53	0.72	0.73	0.40	0.34
PR	0.45	1.58	0.53	0.63	0.40	0.38
SSSP	0.66	2.01	0.72	0.81	0.38	0.34

We evaluate the CAM functionality in GAS using HSPICE circuit level simulations in 45 nm technology. We use VTEAM memristor model [28] for our memory device simulation with R_{ON} and R_{OFF} of $10k\Omega$ and $10M\Omega$ respectively. We also cross-validate the computed energy consumption and performance of the crossbar memory blocks using NVSIM [29]. The architectural parameters of the system simulated in our experiments are shown in Table 2. Due to the overhead of the input buffer, the energy and execution time of the CAM increases at a higher rate for the CAM blocks with more than 4096 rows. Therefore, we limit the number of rows in each CAM block to 4096.

Workloads: In this work, we evaluate the performance and energy consumption of the proposed GAS over three popular kernels on five graphs, including three real world graphs and two synthetic graphs. The detailed descriptions of graphs are shown in Table 3. For each application, we implement algorithms based on the framework introduced in the previous section. We simulate BFS and SSSP entirely while five iterations are simulated in PR.

Baseline: We compare the performance and energy efficiency of GAS with two baseline systems: a system with a state-of-the-art indirect-memory prefetching (IMP) [17] and a system with prefect L1 caches (Ideal). In the IMP system, an indirect memory prefetching mechanism is implemented with a basic streaming prefetcher. The system has 16 prefetching entries and 4 indirect prefetching entries, and the maximum number of prefetched blocks is 16. In the ideal system, we assume all memory accesses can be served in the L1 cache, which means the system has an infinite cache capacity and memory bandwidth. We also test GAS without prefetching mechanism to justify the effect of memristor memory and prefetching. There is no meta-data table in this system, and all memory accesses are directly handled by on-chip buffer and GAS memory.

5.2 Overall Results

Table 4 shows the overall results of running workloads on GAS without prefetching, GAS, and ideal architecture. All results are

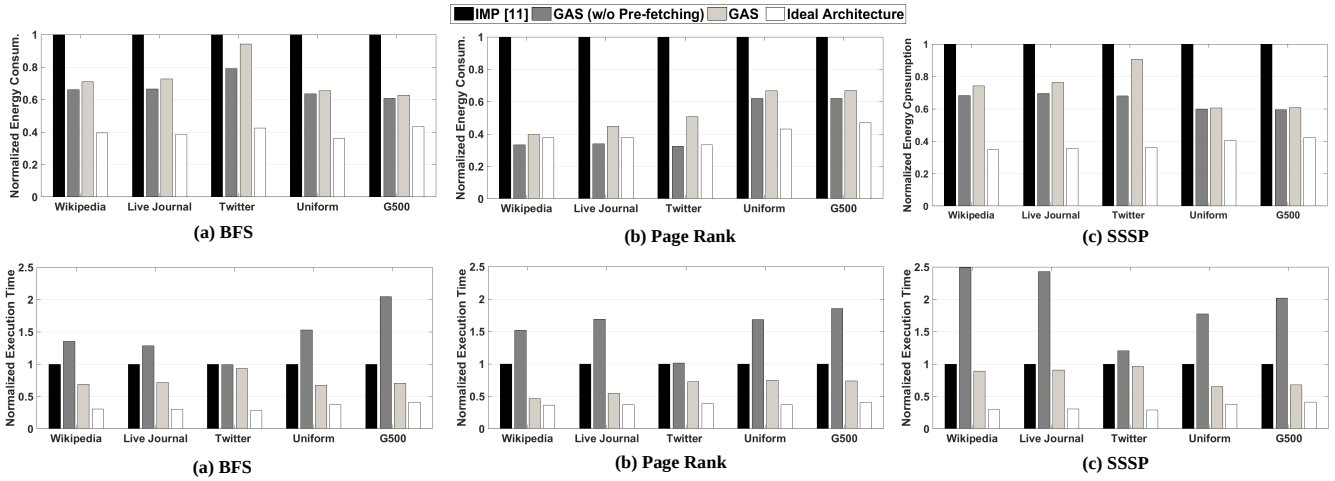


Figure 4: Energy consumption and execution time of different systems over different workloads normalized to IMP.

normalized to the result of the IMP architecture. Our evaluations show that GAS reduces the average energy consumption of BFS, PR, and SSSP applications by 28%, 47%, and 28% respectively. The average reductions in execution times of three applications are 27%, 37%, and 19% respectively. As compared to the result of GAS w/o prefetching, GAS consumes 10% more energy which mainly comes from associative search and writing search results to GAS memory. However, prefetching improves the performance by 2.4× as compared to GAS w/o prefetching. Since PR checks and produces messages for all the edges in the graph during each iteration, the number of messages processed and accessed in each iteration is equal to the total number of edges in the graph. This access pattern leads to many random accesses which dominates the execution of application. In such a case, GAS removes the effect of cache misses introduced by random accesses and further accelerates the application by maximizing bandwidth with prefetching. On the other hand, GAS has the least improvement over SSSP which has a lot of accesses to edge information, like weights. The performance bottleneck of SSSP is due to accesses to messages as well as edge information. Furthermore, each vertex in SSSP is activated in multiple iterations which reduces the number of incoming messages for each vertex. Less message for a vertex reduces the benefits of accurate prefetching. However, GAS still decreases the energy consumption and execution time of SSSP by 28% and 19% respectively.

5.3 Performance and Energy Comparison

Figure 4 shows the detailed results of the energy consumption and execution time of five graphs over different architectures. Our evaluations show that all applications provide less efficiency on real-world graphs (*Wikipedia, Live Journal* and *Twitter*), as compared to synthetic graphs (*Uniform, G500*). This is caused by the fact that real-world graphs are commonly less random than synthetic graphs. For instance, articles about a common topic in the Wikipedia are represented by continuous indexes and refer to each other. Therefore, both incoming and outgoing edges to such vertices are stored together. In contrast, synthetic graphs are generated randomly based on a pre-defined degree of distribution over all vertices, so they have more irregular structures and thus more random accesses.

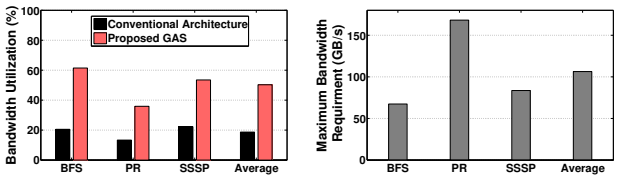
Over all applications, the ideal architecture provides 69.8% and 71.4% reduction in execution time and energy consumption respectively, as compared to the IMP architecture. Although the ideal architecture provides the maximum efficiency in most cases, GAS can still perform very close to the ideal architecture for some workloads, incurring a performance loss within 5%. This is because the on-chip buffers in GAS significantly reduce the overhead of random message accesses, which contributes significantly to the energy consumption and execution time in those specific cases. For the ideal architecture, even though all memory accesses are served by L1 cache, each memory access still needs to look up the cache tag and access one cache block. The size of this access can sometimes be larger than the requested data itself. The comparison between GAS and GAS w/o prefetching shows that the improvement brought by prefetching is less in *Twitter* than other graphs. This is caused by the low average degrees for vertices. In this case, the number of messages sent to each vertex is small which is hard for prefetching to accelerate.

5.4 Memory Access Behavior

GAS improves the performance of the system by offloading random memory accesses to a specific memory system. The performance difference between GAS and the IMP architecture mainly comes from different memory access behaviors. Table 5 lists the results of L1 cache miss rate and average memory access latency of different applications running on GAS and the IMP architecture. The results show that the miss rate and memory latency are consistent with those reported in Figure 4. For example, performance improvement of GAS on PR comes from lower cache miss rate and average memory access latency. For SSSP, reduction in cache miss rate cannot provide similar latency reductions. This is due to a large number of memory accesses to edge information (weights) in SSSP, which cannot be handled by GAS. The loaded edge information cannot be kept in cache until the next access (in next iteration) due to its large size. Thus, it causes a large number of cache misses to edge information which introduces long memory access latency. However, the cache miss rate is not increased significantly by accessing edge information because many continuous edges are stored in the same cache block.

Table 5: Memory access behavior of the IMP architecture and GAS over different workloads.

		BFS					PR					SSSP				
		Wiki	**LJournal**	**Twit**	**Uniform**	**G500**	**Wiki**	**LJournal**	**Twit**	**Uniform**	**G500**	**Wiki**	**LJournal**	**Twit**	**Uniform**	**G500**
IMP Architecture	*Mem latency (ns)*	5.67	5.68	4.85	4.62	4.19	5.74	5.64	7.72	3.90	3.67	6.27	6.05	4.95	4.51	4.37
	Cache miss (%)	8.11	7.22	5.42	4.61	4.45	10.21	8.44	9.73	3.53	3.48	9.63	8.12	5.51	4.46	4.62
GAS	*Mem latency (ns)*	3.72	3.98	4.55	2.06	2.09	1.15	1.32	2.23	1.89	1.77	3.85	3.87	4.26	2.11	2.03
	Cache miss (%)	4.70	4.80	5.01	2.04	1.95	1.00	1.40	2.40	1.30	1.24	6.30	6.12	4.70	1.99	2.11



(a) (b)

Figure 5: (a) Average bandwidth utilization (b) Maximum bandwidth requirement.

Figure 5a shows the bandwidth utilization during *Gather* phase of each application on GAS and the IMP architecture. The maximum bandwidths of the DRAM chip and GAS are kept the same for a fair comparison. The bandwidth of GAS is supposed to be fully utilized by prefetching unless there is no future message to be fetched for the current vertex.The result shows that GAS improves the average bandwidth utilization of BFS, PR, and SSSP by 3.0×, 2.7×, and 2.4× on average, respectively over all workloads by prefetching messages at the maximum bandwidth. Figure 5b shows the maximum bandwidth requirements averaged over different applications. For instance, the PR requires the maximum bandwidth of 168.1 GB/s, which means that we can further improve the GAS efficiency by providing higher bandwidth between GAS and processing cores.

6 CONCLUSION

We propose GAS, a heterogeneous memory architecture to accelerate graph processing applications. GAS removes random memory accesses in graph processing applications, by utilizing associative search to infer the exact memory access pattern while accessing random messages in the vertex program model. We proposed a hardware prefetching mechanism to improve the performance of GAS and an interface to integrate GAS with different applications. We simulate the performance of GAS on three popular graph kernels over five well-known graphs. The experimental results show that GAS can significantly reduce cache misses and improve the bandwidth utilization of conventional architecture with a state-of-the-art prefetching mechanism. These enhancements result in significant improvements on energy and performance of graph applications.

ACKNOWLEDGMENT

This work was partially supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and also NSF grants #1730158 and #1527034.

REFERENCES

[1] Y. Low *et al.*, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.
[2] G. Malewicz *et al.*, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010.
[3] S. Beamer *et al.*, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *IEEE IISWC*, pp. 56–65, IEEE, 2015.
[4] T. J. Ham *et al.*, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *IEEE/ACM Micro*, pp. 1–13, IEEE, 2016.
[5] M. M. Ozdal *et al.*, "Energy efficient architecture for graph analytics accelerators," in *ACM/IEEE ISCA*, pp. 166–177, IEEE, 2016.
[6] N. Sundaram *et al.*, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.
[7] M. Imani *et al.*, "Ultra-efficient processing in-memory for data intensive applications," in *ACM DAC*, p. 6, ACM, 2017.
[8] M. Imani *et al.*, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *IEEE ASP-DAC*, pp. 757–763, IEEE, 2017.
[9] M. Imani *et al.*, "Nvquery: Efficient query processing in non-volatile memory," *IEEE TCAD*, 2018.
[10] M. Imani *et al.*, "Efficient query processing in crossbar memory," in *IEEE ISLPED*, pp. 1–6, IEEE, 2017.
[11] J. Sim *et al.*, "Lupis:latch-up based ultra efficient processing in-memory system," in *IEEE ISQED*, pp. 1–6, IEEE, 2018.
[12] S. Salehi *et al.*, "Mitigating process variability for non-volatile cache resilience and yield," *IEEE Transactions on Emerging Topics in Computing*, 2018.
[13] N. Khoshavi *et al.*, "Variation-immune resistive non-volatile memory using self-organized sub-bank circuit designs," in *IEEE ISQED*, pp. 52–57, IEEE, 2017.
[14] L. Nai *et al.*, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *IEEE HPCA*, pp. 457–468, IEEE, 2017.
[15] L. Song *et al.*, "Graphr: Accelerating graph processing using reram," *arXiv preprint arXiv:1708.06248*, 2017.
[16] S. Ainsworth *et al.*, "Graph prefetching using data structure knowledge," in *Proceedings of the 2016 International Conference on Supercomputing*, p. 39, ACM, 2016.
[17] X. Yu *et al.*, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 178–190, ACM, 2015.
[18] M. S. Riazi *et al.*, "Camsure: Secure content-addressable memory for approximate search," *ACM TECS*, vol. 16, no. 5s, p. 136, 2017.
[19] M. Sharad *et al.*, "Ultra low power associative computing with spin neurons and resistive crossbar memory," in *ACM DAC*, p. 107, ACM, 2013.
[20] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *IEEE HPCA*, pp. 445–456, IEEE, 2017.
[21] M. Imani *et al.*, "Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing," in *IEEE/ACM DATE*, pp. 373–378, EDA Consortium, 2016.
[22] M. Imani *et al.*, "Approximate computing using multiple-access single-charge associative memory," *IEEE TETC*, 2016.
[23] T. E. Carlson *et al.*, "An evaluation of high-level mechanistic core models," *ACM TACO*, vol. 11, no. 3, p. 28, 2014.
[24] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, pp. 190–200, ACM, 2005.
[25] S. Li *et al.*, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 469–480, IEEE, 2009.
[26] W. Heirman *et al.*, "Power-aware multi-core simulation for early design stage hardware/software co-optimization," in *IEEE PACT*, pp. 3–12, IEEE, 2012.
[27] P. Shivakumar *et al.*, "Cacti 3.0: An integrated cache timing, power, and area model," 2001.
[28] S. Kvatinsky *et al.*, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015.
[29] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory," in *Emerging Memory Technologies*, pp. 15–50, Springer, 2014.
[30] P. Boldi *et al.*, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *ACM WWW*.
[31] P. Boldi *et al.*, "The WebGraph framework I: Compression techniques," in *ACM WWW*, (Manhattan, USA), pp. 595–601, ACM Press, 2004.
[32] S. Beamer *et al.*, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.
[33] "Graph500 benchmark." http://graph500.org/.