ExtraV: Boosting Graph Processing Near Storage with a Coherent Accelerator

Jinho Lee[†], Heesu Kim[*], Sungjoo Yoo[*], Kiyoung Choi[*],
H. Peter Hofstee[†,‡], Gi-Joon Nam[†], Mark R. Nutter[†], and Damir Jamsek[†]

[†]IBM Research [*]Seoul National University [‡]TU Delft

[†]{leejinho, hofstee, gnam, mrnutter, jamsek}@us.ibm.com
[*]hkim@dal.snu.ac.kr, sungjoo.yoo@gmail.com, kchoi@snu.ac.kr

ABSTRACT

In this paper, we propose ExtraV, a framework for near-storage graph processing. It is based on the novel concept of *graph virtualization*, which efficiently utilizes a cache-coherent hardware accelerator at the storage side to achieve performance and flexibility at the same time. ExtraV consists of four main components: 1) host processor, 2) main memory, 3) AFU (Accelerator Function Unit) and 4) storage. The AFU, a hardware accelerator, sits between the host processor and storage. Using a coherent interface that allows main memory accesses, it performs graph traversal functions that are common to various algorithms while the program running on the host processor (called the host program) manages the overall execution along with more application-specific tasks. Graph virtualization is a high-level programming model of graph processing that allows designers to focus on algorithm-specific functions. Realized by the accelerator, graph virtualization gives the host programs an illusion that the graph data reside on the main memory in a layout that fits with the memory access behavior of host programs even though the graph data are actually stored in a multi-level, compressed form in storage. We prototyped ExtraV on a Power8 machine with a CAPI-enabled FPGA. Our experiments on a real system prototype offer significant speedup compared to state-of-the-art software only implementations.

1. INTRODUCTION

Large-scale graph processing is one of the key applications in big data analytics for various domains such as social networks, biomedical problems, and even VLSI designs. At the same time, the volume of data is growing at an exponential rate. According to [4], Facebook users share nearly 2.5 million pieces of contents per minute while email users send over 200 million messages every minute. The explosion of information results in large scale graph databases and creates a need for a new kind of system that can process extremely large graphs with a reasonable throughput.

Thus far, the scalability issue of graph size has been mainly addressed by a cluster of servers [40, 42, 24]. However, solutions based on clustered servers are known to suffer from scaling problems due to sub-optimal partitioning and IO complexity [37]. Recently, out-of-memory graph processing has become popular. Out-of-memory systems usually use a single server machine with large storage devices (such as an SSD (Solid-State Drive)) that stores the graph data [37, 46, 29, 55]. Earlier research has demonstrated that such systems are promising alternatives to cluster-based approaches. However, out-of-memory approaches fundamentally suffer from low disk bandwidth. Thus, many approaches are based on partitioning graphs into chunks, thereby exploiting the high sequential bandwidth of disks. The partitioning incurs the overhead of converting data between disk format and main memory format. The conversion overhead often occupies a significant portion (88% in [37]) of total runtime.

In conjunction with the problem, *Near-Data Processing* (NDP) seems to be a promising solution for out-of-memory graph processing. By processing buffer management and traversals at the storage side, a significant portion of the conversion overhead can be saved, and the corresponding storage bandwidth demands can be mitigated. NDP is drawing significant attention from the industry these days in many ways. A direct example would be accompanying SSDs with SQL processing engines [22] or programmable components [47, 25]. However, offloading all of the graph processing to the storage side can be inefficient due to the limitation of the computing power and memory bandwidth/capacity of the processing logic at the storage side.

In this work, we provide further improvements for out-of-memory graph processing in the direction of near-data processing by leveraging coherent accelerator interfaces. Coherent accelerator interfaces [48, 27] allow processors and accelerators to share information in main memory in a seamless way. It is considered one of the most important technologies to maintain the growth of computing power. By placing a coherent accelerator at the storage side, we can exploit the benefit of NDP while not giving up the use of the flexibility of CPUs and the system main memory.

We propose the *ExtraV* framework (that **extra**cts and

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

traverses graphs). In the framework, a coherent accelerator is placed in front of the storage device and communicates with the processor and its main memory. The common library functions for general graph processing are implemented and executed at the storage side by the accelerator unit to provide speedup, while algorithm-specific tasks are left to the host processor so that the users can freely implement their own algorithm in a flexible way.

For efficient coordination of the coherent accelerator, we devised a novel scheme of *graph virtualization*. By using graph virtualization, we provide the program running on the host processor with high-level abstractions that the graph in the storage resides in main memory in the order that the program wants to process it. Upon the host program's request, the accelerator of ExtraV accesses the graph data in the storage, interpret the data, and bring it to the main memory in time by exploiting the feature of coherent accelerator interface. Therefore the host program can digest the data from memory without the overhead of directly accessing the storage device, unlike sharding techniques, while the actual format of the data or computations done in the accelerator is invisible to the host program.

Additionally, because of the virtualization, the accelerator can apply optimizations to the data under the abstraction layer, without affecting processor functions or programmability. We propose two optimization schemes, (1) expand-and-filter and (2) multi-versioning.

Expand-and-filter first decompresses (i.e., expands) graph data fetched from the storage and sends only necessary data to the host processor by filtering out the unnecessary portion of the data, which improves effective bandwidth between the disk (to be exact, the hardware accelerator) and the host processor thereby enhancing the performance of graph processing. We also design a randomly accessible, hardware-friendly compression scheme suited for expand-and-filter. On the other hand, multi-versioning allows past states of the graph to be visible, and enables tracking. It also provides an efficient interface for making modifications to the compressed graph without having to decompress the current data.

Our contributions are summarized as follows:

- Proposing *ExtraV*, a framework using a coherent accelerator and a graph-specific compression scheme to boost near-storage graph processing.
- The design of graph virtualization, a programming model that allows designers to focus on algorithm-specific functions while implementation details (e.g., (de)compression, filtering, multi-versioning, etc.) are hidden and handled by the coherent hardware accelerator.
- Proposing an *expand-and-filter* technique to amplify the effective bandwidth of storage devices.
- Proposing a compression scheme that is suitable for enhancing the performance of graph processing.
- Hardware-managed multi-versioning that allows change tracking and non-invasive updates.
- The design of a hardware accelerator that processes a compressed graph and feeds it to the host processor using a coherent interface.

2. RELATED WORK

Near-data Processing The idea of near-data processing was once popular in the late 90's. It is now seeing a resurgence due to some key enabling technologies emerging. While there is also numerous related work on applying NDP (also called *processing-in-memory*) for memories such as DRAM [38, 12, 52, 39] or STT-MRAM [26], in-storage processing is also getting more attention at the industry side because emerging storage devices such as SSDs provide a good environment for their implementations.

SmartSSD [22] is one of the early implementations of in-storage processing, which places a subset of an SQL server in the SSD controller's firmware. Willow [47] explored through programmable interface for SSDs to support in-storage processing. Recently, [25] proposed a framework for general in-storage processing for SSDs.

BlueDBM [34] and Tesseract [11] are interesting architectures that exhibit similar structure in different scales. BlueDBM is a rack-scale system where each node has flash devices and in-storage processors. Tessaract uses a 3D stacked memory with on top of a logic layer as a building block, and connects multiple such blocks together to form an accelerator.

However, none of these explore the out-of-memory graph processing domain, and the use of coherent accelerators. ExtraV aggressively uses coherent interfaces to bring flexibility and ease of development to the framework.

In-memory Graph Analytics The Pegasus framework [35] is a graph processing system based on MapReduce [21] extending matrix-vector multiplications to graph processing. However, such general distributed processing frameworks were found to be inefficient for graph processing due to irregular access patterns, and consequently many frameworks specifically targeting graph analytics have been presented. Pregel [42] proposed using a Bulk Synchronous Parallel [49] model for distributed graph processing. GraphLab [40] employed a shared memory model instead of a message passing scheme. Its successor, PowerGraph [24] introduced the GAS (Gather, Apply, Scatter) model for graphs with power-law distributions. Most recently, PGX.D [31] demonstrated significant performance improvement by implementing efficient communication and workload balancing.

Out-of-memory Graph Analytics GraphChi [37] is the first out-of-memory system to show the potential to match the performance of clustered servers. GraphChi carefully optimized sequential access patterns to the disks for performance improvement. X-stream [46] uses the edge-centric processing model to minimize random accesses to the disks. Turbograph [29] proposes the pin-and-slide model to take advantage of inherent parallelism available on SSDs. Llama [41] reduces the overhead of buffer management using mmap. Additionally, it introduces the new concepts of snapshots and indirection tables to support multi-version graphs. GridGraph [55] proposes 2D partitioning and selective scheduling to further improve system performance.

More recently, FlashGraph [53] and PrefEdge [43] are architectures that optimize semi-external graph processing on SSDs. FlashGraph is implemented on a user-space file system, and overlaps the latencies of requests to gain high IOPS (Input/Output Operations Per Seconds). PrefEdge is implemented as a graph-specific prefetch engine to improve the throughput to SSDs. However, even with advanced SSDs, the overall system performance is not on a par with the cluster server in-memory graph analytics. Also, to the best of our knowledge, none of the previous approaches try to take advantage of hardware accelerator technology commonly available in recent heterogeneous computing platforms [6].

Hardware based Approaches [30] introduces new implementations of BFS, SSSP, and APSP algorithms on GPUs using CUDA. [32] improves the performance of BFS using hybrid approaches between host processor and GPU. [19] proposes building a computer architecture suitable for irregular memory accesses. Also, [28] and [44] describes accelerators that maximize memory parallelism in graph processing.

All those related works assume in-memory processing, and none of them consider providing a new abstraction to the CPU or amplifying the storage bandwidth. We believe that our approach can be combined with these approaches to further enhance the performance. However, it is beyond the scope of this paper.

3. BACKGROUND: CAPI (COHERENT ACCELERATOR PROCESSOR INTERFACE)

Coherent accelerations are an emerging technology that draws tremendous interest from both industry and academia as a new heterogeneous computing platform for using accelerators in a system. IBM CAPI [48] and intel HARP [27] are two available well-known systems. This work makes use of IBM CAPI (Coherent Accelerator Processor Interface).

CAPI is a new feature introduced in the IBM POWER8 processor. It is intended to assist the host POWER8 processor by offloading some workloads to more efficient hardware accelerators such as FPGAs or GPUs. One unique feature of CAPI is that it enables accelerators to coherently access the entire main memory of the system. The processor and accelerators share the same memory space and virtual addresses while the cache coherence is supported by hardware. This is achieved by CAPP (Coherent Attached Processor Proxy) implemented in a host processor and PSL (POWER Service Layer) implemented on an accelerator unit. The CAPP maintains a cache directory of shared memory space. The PSL works in concert with the CAPP unit across a PCI-e connection. It provides a straightforward interface to the accelerator to grant access to coherent memory. When needed, the CAPP and PSL communicate with each other for memory coherent actions such as writebacks or cache reads. Because all the data movement and coherency are managed by the PSL and CAPP, the client can focus on their accelerator algorithms and implementations.

The AFU (Accelerator Function Unit) is the place where custom acceleration functions are implemented and executed. An accelerator consists of one or more AFUs. Each AFU can be customized with application specific user-defined logic. The PSL takes charge of the link between the AFU and the host system, and it is comprised of several interfaces including command interface, buffer interface, response interface, control interface, and MMIO interface. The *command interface* executes memory commands issued from the AFU. It can support various options such as bypassing PSL and CAPP caches, or modifying initial cache line state according to the scenario of future accesses. All data are moved through the *buffer interface*. The PSL reorders the data move commands to improve memory operation latency. Since the order of responses to memory operations is not guaranteed, it sets a tag to each command to identify them. The *response interface* notifies the completion of commands to AFU, along with the associated credits, so that the AFU can continue issuing commands to the PSL. The host can start or reset the AFU via the *control interface*. Lastly, the *MMIO interface* maps the AFU registers and allows a host program to read or write AFU registers directly via handshaking.

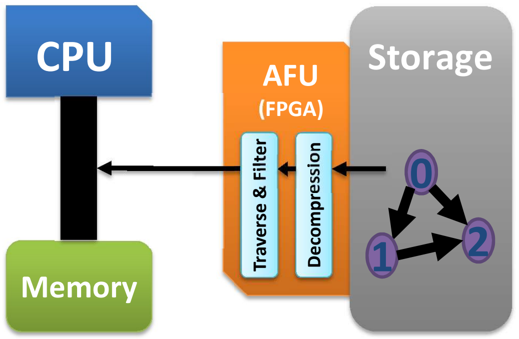

Figure 1: Overview of the ExtraV framework.

Because CAPI can remove the need for explicitly copying the data between processor and the accelerator, one can expect performance advantage with low communication latencies. IBM has built a key-value store using CAPI in support for NoSQL [16], and [23] has implemented accelerating arithmetic kernels using CAPI. In this work, we aggressively utilize the fine-grained communication capability of CAPI to boost graph analytics performance.

4. THE EXTRAV FRAMEWORK

4.1 Framework Overview

Figure 1 shows an overview of the ExtraV framework. The system has four main components: CPU, main memory, AFU, and storage.

The AFU is placed in front of storage and processes the common work that is independent of any graph processing algorithm. For instance, assume that a graph processing program running on the CPU tries to retrieve an edge list consisting of the vertices satisfying a certain condition. The host program sends a request to the AFU. Upon the host program's request, the AFU first interprets the graph data for the requested vertex, and then retrieves the corresponding edge list data from the storage. When the edge list data arrives from the storage, it decompresses the edge list, applies filtering if needed, and transfers the results to the main memory, so that the host program sees the edge list on the memory, in the order that it wants to process. All this work is invisible to the host program. Hence the graph processing program running on the host processor does not need any knowledge of the work done by the AFU, or the format in which the data is stored in the storage. By placing the AFU near storage, ExtraV not only offloads compute-intensive workloads to hardware, but also reduces the off-chip data traffic. While the same functionality could also be implemented as a software driver, this would cause the unfiltered data to travel from storage to memory and bring the computations back to the CPU with loss of performance.

In this work, we assume a semi-external graph processing, where the main memory cannot store the entire graph but can store the attributes of vertices. Since the number of edges in a real-world graph is often far (more than 10×) larger than

Table 1: Streaming Queries Used for Graph Virtualization

Type	Input	Output
Startup	G (identifier with level)	num_vertices(G), num_edges(G)
Sequential	In/Out, F⊂G (optional)	{N|N=nbr(v), v∈G∩F}
Job-queue	In/Out, Q⊂G, F⊂G (optional)	{N|N=nbr(v), v∈Q∩F}
Degree	In/Out	{d|d=degree(v), v∈G}

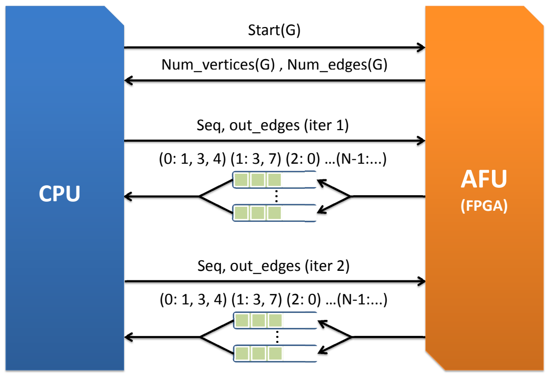
Start(G)
Num_vertices(G) , Num_edges(G)
Seq, out_edges (iter 1)
(0: 1, 3, 4) (1: 3, 7) (2: 0) ...(N-1:...)
CPU
AFU
(FPGA)
Seq, out_edges (iter 2)
(0: 1, 3, 4) (1: 3, 7) (2: 0) ...(N-1:...)

Figure 2: Graph virtualization programming model.

that of vertices [8], this is an acceptable assumption, and this approach has been adopted by existing works [53, 43]. We leave the extension to the environment where the vertices do not fit into the main memory as future work. The stored graph data is in the *CSR* (Compressed Sparse Row) form, but the edge list is further compressed (See Section 4.4).

The main memory stores temporary data and vertex attributes. Examples would be parent values in the BFS algorithm or PageRank values for individual vertices in the PageRank algorithm. Optionally, the memory also stores some values that are shared between the processor and the AFU using the coherent interface. As will be explained later, the job queue and the filter bitmap are two items that are shared between the two.

The program running on the CPU processes the application-specific part of the algorithms. For example, it can propagate the PageRank value of a vertex to its neighbors or check to see whether a certain vertex is already visited or not. Although the application-specific part can also be executed in an AFU, accessing the vertex attributes on the main memory will consume a lot of bandwidth. Since the bandwidth available to AFU is significantly less than that of the host processor (about 3GB/s for PCI-e), processing such large data in the main memory is much more efficient at the host processor side. Furthermore, designing custom hardware for application specific algorithms would require significantly large hardware as well as design effort. Thus, by leaving the application-specific code to the host processor, any new algorithm or improvements of the existing algorithm can be implemented easily by software and we gain both algorithmic flexibility and memory access efficiency.

4.2 Programming Model

We propose graph virtualization as a high-level abstraction of graph processing on the ExtraV system. We need a high-level abstraction mainly because the round-trip latency between the processor and the AFU is too long (about 1 µs on PCI-e Gen3 x8). If such fine-grained communications are exposed to graph processing programs running on the host processor, the communication overhead can become prohibitively expensive. For instance, in the case of PageRank, if each access to an entry in the edge list is performed in the fine-grained manner incurring 1 µs of latency, the performance of graph processing will be extremely poor. The graph virtualization programming model tries to minimize such communication overhead and thus maximize the algorithm speedup.

Figure 2 shows the graph virtualization model. For the start-up procedure, the host program informs the AFU of the graph it intends to process, and as an acknowledgement, the AFU sends the metadata (the numbers of vertices and edges). Then, instead of requesting individual vertices or edges, the program sends a *streaming query* to the AFU, e.g., for a specific set of edges. When the AFU receives the query, it starts sequentially streaming the response, e.g., the set of edges, to the host program. Each response to the query is written to the response queues in the main memory in the form of (vertex: {neighbor list}), and the responses are continuously sent (e.g., until all the edges are traversed). These response queues are managed by leveraging the coherent interface. Because of this, graph virtualization does not pay expensive overheads such as AFU polling on memory, CPU polling on MMIO registers or issuing hardware memory barriers, which exist in conventional queue-pair models used for conventional accelerators. Instead, the CPU and the AFU spin on their own caches and get informed by the coherence mechanism. If the graph processing algorithm on the CPU wants to perform next iteration, the host program sends the streaming query again. For algorithms that have phases, the type of streaming query may differ depending on the specific phase (as will be explained later).

In our implementation, there are four streaming query types as shown in Table 1: *startup, sequential, job-queue, and degree.* In the table, G represents a graph, with a version specified. F and Q are the subsets of a graph which are used to represent vertices in the filtering set and the job-queue, respectively. $nbr(v)$ represents the neighbors of vertex v, and $degree(v)$ represents the degree of the vertex v. Startup queries tell the AFU which graph at which level (for multi-versioning. See Section 4.5) the host program wants to process, and the AFU responds with the number of vertices and edges in the graph. Sequential queries are used for many iteration-based graph analytics that access all vertices in an iteration and returns their neighbor lists. Job-queue queries are used for algorithms such as BFS and Dijkstra which require random accesses to a subset of vertices. The host program writes the list of vertices to be processed in a job queue Q and the AFU returns to the host program the neighbor lists of the vertices retrieved from the job queue. The job queue also resides in the memory and can be accessed by both the host program and the AFU. Lastly, the degree queries are used for some algorithms that require degrees of

Table 2: Compression format used in ExtraV

Vertex

Level (10 bits)	Offset to edges (54 bits)

Edges

Num_intervals	Residual_offset	{ ($\Delta begin_1, len_1$), ...}	{ $\Delta residual_1, ...$}	EOL/CONT

VLI (Variable Length Integer)/Control Tokens

Len/control (6 bits)			Additional
000000-111101	VLI length	Bit length of the following integer	Binary encoded integer
111110	CONT	Continuation reference	Level (VLI), Offset (VLI)
111111	EOL	End-of-list	-

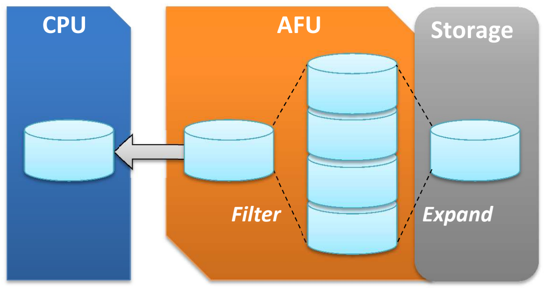

Figure 3: An illustration of the expand-and-filter scheme.

vertices. The degree query is used typically at the first phase of an algorithm, and the degrees are stored in the memory for the remaining phases. Optionally, the streaming queries can provide a filter bitmap F to implement the expand-and filter mechanism (see Section 4.3). These types of streaming queries were enough for our benchmark algorithms, but they can be easily extended to include other types if needed.

The use of the streaming query provides virtualization of graphs to the programs running on the processor. In the host program's view, entries in the response queues are visible, and the program needs to sequentially read the contents in the response queues in an illusion that the graph data is pre-ordered in the way that it is going to be accessed. Under the graph virtualization programming model with streaming queries, each iteration requires only one round-trip latency, which is only a small fraction of the total processing time for an iteration (ranging from a few seconds to a few minutes). There is an additional round-trip overhead at the start-up procedure, but it is required only once for the entire run, and the latency is amortized over all the iterations. Note that when the host program reads the responses, they are deleted from the main memory, so the response queues require a small amount of memory space (64MB in our experiments).

4.3 Expand-and-filter

In most out-of-memory graph processing frameworks, the system bottleneck is the bandwidth of the storage device. Taking advantage of graph virtualization, ExtraV uses the expand-and-filter scheme under the abstract layer to effectively amplify the storage bandwidth.

Figure 3 illustrates the expand-and-filter scheme. In the system, there are two inter-device interfaces that can become the performance bottleneck: the storage-AFU interface, and the AFU-processor interface. First, in order to reduce the bandwidth consumption of the storage-AFU interface, graph data is stored in a compressed form in the storage device (See Section 4.4). When the AFU reads the graph data, it decompresses the data on-the-fly (expand). The provided bandwidth of the storage device is effectively amplified by the compression ratio.

However, when the data is decompressed, the data becomes a few times larger than the original compressed data, and this increases the bandwidth requirement of the AFU-processor interface. Depending on the bandwidth ratio between the AFU-processor interface and the storage-AFU interface, this may become a new bottleneck to the system. To mitigate this effect, the AFU can drop some of the data that is not going to be used by the host program (filter). Some algorithms such as BFS [13] and betweenness centrality [17] maintain a bitmap that determines which neighbors actually participate in processing. If the filtering option of the streaming query is set, the AFU looks up a shared bitmap, and gives as output only the neighbors of the vertices whose bits are set in the shared bitmap. The bitmap resides in the memory like the job queue, and is shared between the AFU and the host program using a coherent interface. As a result, the bandwidth requirement of the AFU-processor interface can be greatly reduced.

4.4 Compression Scheme

Compression is one of the key features in ExtraV that delivers a speed superior to existing frameworks. A good compression scheme for graph processing requires three characteristics: high compression ratio, random accessibility, and fast decompression speed. The compression ratio determines how much bandwidth amplification can be obtained from the storage. However, it is also important to have a high enough decompression speed to exploit the full bandwidth of the storage. Otherwise, the storage would be underutilized and the speedup would be reduced. Random accessibility is also essential to support graph processing algorithms (e.g., the job-queue query).

The BV (Boldi-Vigna) scheme [15] provides a high compression ratio for web graphs, but to decompress a vertex, it has to repeatedly follow backward dependencies and decompress many other vertices. This would not only slow down the decompression speed, but also increase the amount of data the AFU has to fetch from the storage. Also, BL (Back-Links) compression [20] stores only one side of the bidirectional edges, and thus it is almost impossible to access a random vertex. Pathgraph [51] adopted some ideas based on variable length integers, but their technique is simple and the compression ratio was not high enough for our purposes. Our compression scheme adopts some key ideas from BV and

```
master_thread:
    kernel->init()
    do {
        kernel->prepare()
        streaming_query(kernel->query_type())
        while(num_processed < kernel->worksize())
    } while (!kernel->end() && ++iter < max_iter)
```

(a) Master thread

```
worker_thread:
    while(1) { //worker never returns
        (u, u_nbr) = resp_queue.next()
        kernel->body(u,u_nbr)
        num_processed++       //atomic
    }
```

(b) Worker Thread

```
Procedure PageRank(graph G) :
    init()
    do {
        PR = PR_next
        for u in G {
            for v in in_nbr(u) {
                sum += PR[v] / degree[v]
            }
            PR_next[u] = (1-d)/N + d*sum
        }
        diff += |PR_next - PR| //atomic
    } while (diff > e && ++iter < max_iter)
```

(c) PageRank Algorithm (Pesudo-code)

```
Class PageRank : inherit Kernel
    Procedure prepare() :
        PR = PR_next
    Procedure query_type() :
        return (sequential, in_edges)
    Procedure worksize() :
        return G.num_vertices
    Procedure end() :
        return (diff <= e)
    Procedure body(u, u_nbr) :
        for v in u_nbr {
            sum += PR[v] / degree[v]
        }
        PR_next[u] = (1-d)/N + d*sum
        diff += |PR_next - PR| //atomic
```

(d) PageRank Kernel Implementation of ExtraV

Figure 4: Execution model and the PageRank example.

BL, but ours allows random access and the decompression procedure is hardware-friendly, yet the compression ratio is high enough to provide a significant speedup.

Our compression scheme is mainly based on run-length and differential coding. Using the two coding techniques, the size and the number of data to be stored are reduced. Then variable length encoding is used to achieve additional compression. Table 2 displays the compressed graph format (a vertex and the list of edges connected to the vertex; each edge is represented by a neighboring vertex). The compressed edge list is composed of two parts: intervals and residuals. The interval part stores sets of consecutively numbered (neighboring) vertices, and the residual part stores the remaining neighbors that do not fall into the interval part. The compressed edge list first has the number of intervals and the offset to the beginning of the residuals. The interval parts are in the form of (Δbegin, length). Δbegin stores the difference between the ids at the beginning of the interval and at the end of the previous interval (for the first interval, the difference with the vertex id). For example, if the vertex id is 3, Δbegin of the first interval is 4, and the length of that interval is 5, then the first interval represents neighboring vertices with ids from 7 to 11. We can decompress the residuals by regarding them as intervals with the implied length of zero. Because we have added the residual offset at the beginning of the list, the acceleration hardware can jump to the residuals without decompressing the whole interval part, enabling parallel decompression. All the integers in the compressed edge list are further compressed by using variable-length integer coding. The first six bits represent the length of the binary coded integer, followed by the actual binary number. The code $111111_{(2)}$ and $111110_{(2)}$ are reserved for the end-of-list indicator, to avoid decoding junk (see Section 5), and for continuation to next level indicator (see Section 4.5), respectively.

The vertex list provides additional opportunity for compression with similar techniques. However, we chose not to compress it, because it would incur more overhead in the AFU. Furthermore, the benefit will be small since the vertex list takes only a small portion of the graph data.

4.5 Multi-versioning

Multi-versioning allows graphs to store past versions at different time points. It is not only useful for looking at the history of changes, but also essential to CSR-based graph databases for keeping updates. Addition of new edges or vertices to CSR-based graphs is almost impossible because CSR is an array-based architecture and thus multi-versioning is used. ExtraV provides abstract multi-versioning based on a scheme similar to Llama [41] with a few minor modifications.

In Llama, the edgelist of each version only keeps the delta from the previous version. The end of the edgelist of a version stores a *continuation reference* to the edgelist from the previous edgelist (version and offset) so that reading through the references would eventually provide the whole list of the edges. Each vertex table entry has a reference to the neighboring edgelist with the length of the edgelist. The length information is to know the end of the edgelist and interpret the data as a reference instead of an edge. To avoid the vertex tables in different versions having similar contents, the vertex tables are divided into large chunks called *pages* and only the pages that have changes are newly written. Additionally, each version provides an array of pointers to the pages, allowing re-use of the pages of previous versions.

In ExtraV, the version and offset of continuation references are also encoded using variable-length integers. $111110_{(2)}$ in the length field indicates that the next two integers belong to a continuation reference. $111111_{(2)}$ indicates the end-of-list where there are no further entries. Because of this, there is no need to store the length information of the edgelist in the vertex tables, and thus ExtraV can allocate 64bit for each entry in the vertex table, compared to 128bit in Llama.

4.6 Execution Model

The execution model of ExtraV provides flexibility to the design of custom algorithms. As we assume a semi-external

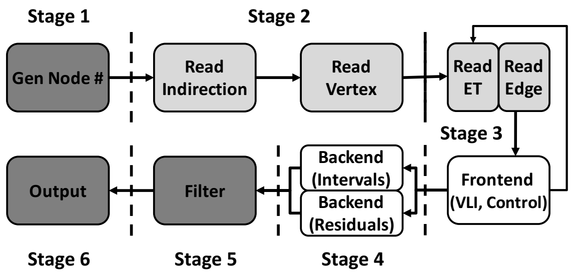

Figure 5: Pipeline architecture of ExtraV workers.

environment, it is intuitive to use a vertex-oriented model as in prior work [53, 43]. The execution model is shown in Figure 4. The backbone software structure of ExtraV consists of a master and a few worker threads, as shown in Figure 4 (a) and (b), respectively. The master thread handles the global jobs which have to be done between iterations (line 4), and invokes streaming queries to the AFU (line 5). After the streaming query is sent, it waits until the worker threads on the host program process the data from AFU responses. When the number of processed vertices reaches the defined work size, the current iteration is ended (line 6). The master then checks for the termination condition (line 7). If it does not meet the condition yet, it proceeds to the next iteration repeating the loop of lines 3-7.

The worker threads wait on the response queue for the AFU to send responses (line 10). Upon receiving a response, it runs the body function that has to be performed on each vertex (line 11). Afterwards, it atomically increases the global variable which indicates the total number of vertices processed by the worker threads (line 12). The worker threads wait again on the response queue and keep running until the end of the algorithm.

Figure 4 (c) and (d) shows pseudo-code for the PageRank algorithm and its implementation on ExtraV, respectively. The PageRank runs as follows. After initializing the program (line 15), it starts the iteration. It traverses through all the vertices in the graph, while obtaining their incoming neighbors (lines 18-19). From the collected neighbors, the PageRank value for the next iteration is calculated (lines 20 and 22) and the convergence error is accumulated (line 24). If the convergence error is below a certain threshold or the max iteration count has been reached, the algorithm terminates (line 25).

Users can implement their algorithm by describing a kernel class and its member functions as Figure 4 (d) shows. The prepare() function handles the job between iterations. The query_type() function determines which streaming query to send to the AFU. In PageRank, each iteration performs sequential traversal of the incoming neighbors. To implement more complicated applications, different query types can be used for each iteration. The worksize() function determines the number of vertices to be processed per iteration. It is usually all the vertices in the graph, but in some algorithms, only a subset of vertices are accessed per iteration, and it should be explicitly determined by the function. The end() function determines the termination condition for the algorithm. For example, the PageRank algorithm terminates when the rank values converge, and the BFS algorithm terminates when all the vertices are visited from the source vertex. Finally, the body() function defines the core jobs to be performed on each vertex.

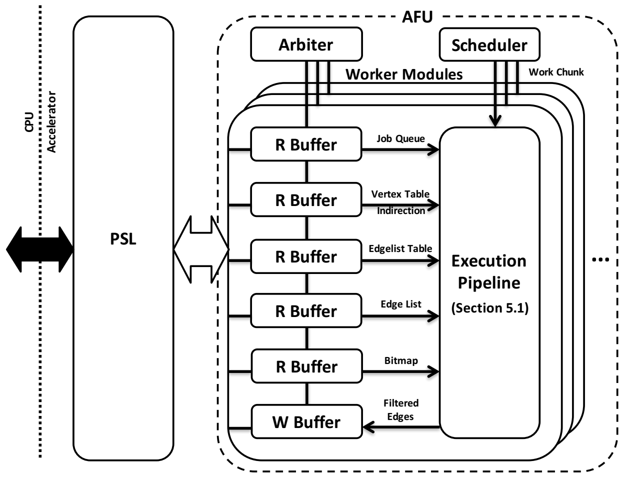

Figure 6: Architecture of the hardware accelerator in ExtraV.

5. ACCELERATOR DESIGN

5.1 Execution Pipeline

Figure 5 shows the pipeline architecture of ExtraV. The boxes shaded in light gray represent modules that read from the storage, while boxes shaded in dark gray represent modules that read/write from the memory. White boxes do not access outside the accelerator.

Stage 1 generates a node number, optionally by reading from the job queue. Stage 2 reads the indirection table to find a page and to see if the node number crosses the page boundary. Then it reads in the corresponding vertex table to find out which edgelist and the offset to look for its neighbors. In stage 3, the edgelist is continuously read, followed by the front-end decompressor that includes VLI (Variable Length Integer) decoding and handling of controls such as length of list, end-of-list, and continuation references. Before reading the edgelist, if the version has been changed from the previously processed vertex, or if a continuation reference is found at the front-end decoder, an entry from ET (edgelist table) is retrieved to find out the location of the edgelist. Stage 4 includes back-end decompression of interval decoding and residual decoding. The results are passed on to Stage 5, where the results are filtered, and only the filtered results are written to the processor memory in Stage 6. We place multiple pipelines into our accelerator design to draw the maximum parallelism.

5.2 Stream Buffer

Even though PSL itself has 256KB of cache for the accelerator, having it shared among all 16 pipelines often causes conflicts that significantly delay the pipelines. To resolve the problem, we place a small buffer called the *stream buffer* on each IO channel of the pipeline. Since most of the accesses to the storage/memory are sequential, we chose the size of each stream buffer to be equal to a single cache-line size (128B). It is the minimum data width that the PSL can access at a time. Those stream buffers not only reduce the cache thrashing by separating IO channels from each other, but also saves latency to/from the PSL cache, functioning similar to a partitioned L1 cache.

While we could use a larger centralized buffer as a cache,

Table 3: Graph Datasets

Graph	Type	#Vertices	#Edges	Max. In-degree	Max. Out-degree
Twitter	Social network	41,652,230	1,468,365,182	770,155	2,997,469
Friendster	Social Network	124,836,180	1,806,067,135	4,223	3,615
MS-ref	Citations	46,742,304	528,682,289	178,438	19,028
Gsh-tpd	Web crawl	30,809,122	602,119,716	2,174,980	526,114

we found that each channel having a single-block sized buffer is more efficient because 1) each channel usually exhibits a sequential access pattern and 2) the data seldom shows temporal locality.

To reduce resource usage, the indirection and vertex table channels share the stream buffer. This does not affect the system much because the indirection is read only once per page (with thousands of vertices) and the two ports are not accessed at the same time.

5.3 Overall Architecture

The design of the hardware accelerator used in ExtraV is shown in Figure 6. The accelerator is designed to work for the CAPI system. At the top level, it consists of a PSL and an AFU as explained in Section 3. The PSL is an interface layer that connects the accelerator to the host memory system. It also maintains coherence between the AFU and the memory. The AFU performs the acceleration jobs. It contains an arbiter, a scheduler, and multiple worker modules.

The scheduler divides the range of work to be done into small chunks, and sends them to the worker modules that are ready to process the next chunks. Upon receiving the work chunk, each worker module starts executing the pipeline with the data read through read-stream buffers. The output data are put to the main memory through the write-stream buffer. Each worker module gets its own response queue, so that they don't suffer from synchronization issues between each other. After a chunk of work is finished, the worker signals the scheduler and the scheduler assigns another chunk to the worker module.

The stream buffers make accesses to outside the AFU; the vertex table, indirections, edgelist table and edgelist are read from the storage, while others access the system main memory. The arbiter is used to avoid conflict between the read/write stream buffers during communication with system objects outside the accelerator.

In addition to relying on parallel processing with multiple worker modules, we employ prefetching to hide the latency of the storage. Inspired by [43], we prefetch data from the storage either in a sequential way, or following the vertices stored in the job queue, according to the type of streaming query being executed.

6. EVALUATION

6.1 Experimental Setup

We prototyped our design on an Alphadata CAPI development card ADM-PCIE-KU3 [1] which has a Xilinx Ultrascale [45] FPGA, and an SSD with a capacity of 512GB and maximum sequential read bandwidth of approximately 540MB/s. The FPGA has 332K CLB blocks and 663K register blocks. The board is connected with the processor using a x8 PCI-express Gen3 interface. The FPGA and the SSD are attached to a CAPI-enabled POWER8 processor [5]. The processor has 20 cores running at 3.7 GHz, each with 512KB L2 cache and 8MB L3 cache. The control group feature in linux is used to limit the physical memory to be 4GB, thus forcing the algorithms to run in a semi-external environment.

The worker modules in the AFU are designed using Vivado-HLS [10] from a C++ model and run at 125 MHz. 16 worker modules are placed inside the AFU, which is enough to draw the full bandwidth out of the SSD attached to the server. The CAPI streaming framework [2] is used to connect the AFU to the PSL. The mmap feature is used to allow access to the SSD from the AFU via the PCI-e channel. Any other interfaces such as SO-DIMM or SATA slots can also be used between the storage device and the FPGA.

We test four real-world graphs, and their statistics are shown in Table 3. Twitter [36], a social network graph, has 42M vertices and 1.5B edges. Friendster [50], another social network graph, has 124M vertices and 1.8B edges. MS-ref [7], a graph of paper citations, has 30M vertices and 0.6B edges. Gsh-tpd [14], representing web crawl graphs, has 39M vertices and 0.94B edges. Four algorithms are tested: PageRank, BFS, average teenage followers and connected components. Table 4 shows the types of streaming queries used for each algorithm. The followings are brief explanations of the algorithms.

AT [33]: AT (Average Teenage followers) iterates over the whole graph, and computes the average number of teenage followers over K years old. AT uses sequential queries.

PR [18]: Being the most famous graph processing algorithm, PR (PageRank) gathers PR values from neighbors of a vertex, and updates the PR value of that vertex. Sequential streaming queries are used to process this algorithm. We ran the algorithm for ten iterations on each graph.

BFS [13]: BFS (Breadth-First Search) visits neighbors in breadth-first order and marks each vertex's parent. It uses job-queue queries and sequential queries with filtering. We implemented the hybrid approach from [13], but the parameters related to transitions between the top-down phase and the bottom-up phase were tuned for our platform. The

Table 4: Test Set Applications

Application	Seq.	Queue	Deg.	Filter
AT	✔			
PR	✔		✔	
BFS	✔	✔	✔	✔
CC	✔	✔		✔

Table 5: Synthesis Results

Module \ Resource	CLB Count	CLB Util	Register Count	Register Util
PSL	82K	30.0%	76K	26.0%
AFU	191K	70.0%	216K	74.0%
↘ Worker (×16)	9.4K	3.4%	13K	4.6%
↘ Ex. Pipeline	5.8K	2.1%	5.6K	1.9%
↘ Stream Buf. (×6)	0.6K	0.2%	1.3K	0.4%
↘ Misc.	41K	14.9%	1.6K	0.5%
Total	273K	100%	292K	100%

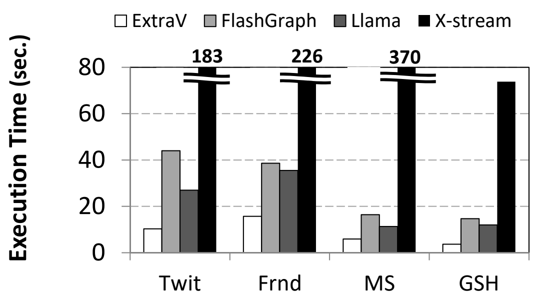

Figure 7: Performance comparison of AT.

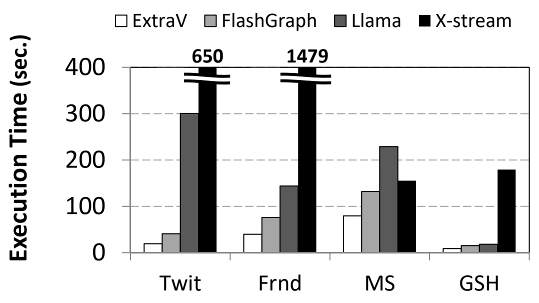

Figure 9: Performance comparison of BFS.

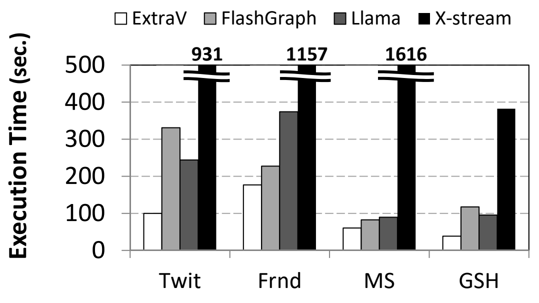

Figure 8: Performance comparison of PR.

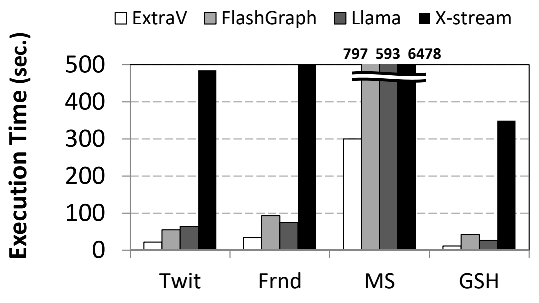

Figure 10: Performance comparison of CC.

algorithm ran until termination.

CC: CC (Connected Components) uses label propagation [54] to group the neighbors of a vertex into a single component. Sequential, filtering and job-queue queries are used. We ran the algorithm until the number of vertices updated reached 1/10000 of the total number of vertices.

6.2 Synthesis Results

The accelerator of ExtraV was synthesized using Xilinx Vivado [9]. The synthesis results of the accelerator are shown in Table 5. The design consumes about 273K CLB blocks and 292K register blocks in the FPGA, which is around 80% and 40% of the FPGA resources, respectively. The PSL, which handles communication and manages coherence with the CPU consumes a little more than a quarter of the FPGA. The AFU is mostly comprised of 16 worker modules, where each worker module consists of an execution pipeline and six stream buffers. The execution pipeline occupies the majority of the CLB blocks in the worker module, while most of its registers are consumed by the stream buffers. Aside from the worker modules, there are a few miscellaneous components in the AFU, which include an arbiter, a scheduler, and some glue logic.

6.3 Performance Results

We compare the performance of ExtraV with three other state-of-the-art out-of-memory graph processing frameworks, Llama [41], FlashGraph [53], and X-stream [46]. We used the same hardware machine environment for all the frameworks. We newly implemented some algorithms that are not provided: the AT on all three frameworks and the CC on Llama and FlashGraph. Figure 7 shows the execution time of the AT (average teenager follower) on four graphs. ExtraV outperforms all the other frameworks. For the twitter graph, ExtraV runs AT in 10.3 seconds while FlashGraph runs in 44 seconds, and Llama finishes in 27 seconds, which translates into 4.27× and 2.62× improvement, respectively. X-stream, on the other hand, suffers from much longer execution time compared to all other frameworks. We found that X-stream dumps its intermediate results to the storage even though there is remaining memory space to hold them. Thus it suffers from extra storage reads and writes, resulting in inferior performance in all benchmarks. The speedup of ExtraV mainly comes from the compression and the reduction of buffer management overhead. Consequently, the speedup of ExtraV in AT over Llama show a strong correlation to the compression ratios of the Graph datasets (See Section 6.6). The geometric mean speedup of ExtraV is 3.29×, 2.47×, and 23.9× over FlashGraph, Llama, and X-stream, respectively.

The performance results of PR (PageRank) in Figure 8 is similar to that of AT. In AT, Llama generally runs faster than FlashGraph because Llama eliminates buffer management overhead by aggressively using mmap. However, FlashGraph performs relatively better in PR, since it adopts selective scheduling, which omits nodes that have already converged. It can be seen that the speedup of ExtraV over FlashGraph for PR is smaller than that of AT. ExtraV performs 2.05×, 2.08×, and 11.2× better than FlashGraph, Llama, and X-stream, respectively.

The performance of BFS (Breadth-First Search) is displayed in Figure 9. Out of the four benchmark algorithms, ExtraV shows the least speedup for BFS. It still outperforms 4.27× over Llama and 14.9× over X-stream. On the other hand, its speedup over FlashGraph for BFS is 1.84×, which is still significant, but far less than those of other algorithms.

What ExtraV is not good at is accessing partial graphs. Even though it provides partial access methods with job queue queries and filtering options, it still suffers from frequent flushes of stream buffers, which slows down the traversal. BFS often requires traversal of significantly large portions of the graphs called frontiers, usually less than a quarter of the entire graph [32]. To traverse in such a way, some

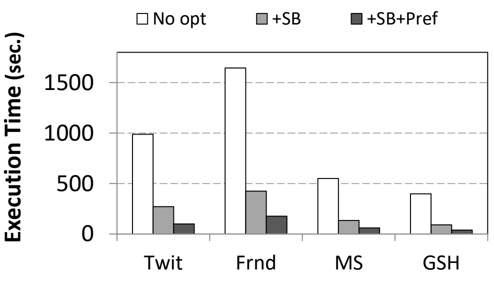

Figure 11: Performance effects of stream buffers and prefetching.

performance is lost during the flushing of stream buffers (as in twitter) or round-trip between CPU and accelerator for frequent streaming queries (as in gsh-tpd). With selective scheduling in FlashGraph, the time to traverse a partial graph is almost proportional to the ratio of the partial graph to the entire graph, explaining the reduction in speedup.

In Figure 10, execution times for CC (connected components) is shown. In CC, each vertex propagates its vertex index, and the neighbors record the smallest as their component index. When a vertex does not get an update, it does not propagate its component index in the next iteration. ExtraV starts with sequential streaming query since all vertices have to participate in the propagation in the beginning. When the number of updated vertices decreases below a threshold, it changes to the usage of job-queue query and processes the remaining small set of vertices. ExtraV runs 2.87× faster than FlashGraph, 2.34× than Llama, and 28.2× than X-stream on the Twitter graph. CC also requires partial traversals in later iterations, but the speedup of ExtraV is huge, unlike BFS. Because the CC algorithm runs asynchronously, the behavior of each iteration is not deterministic, and it can vary greatly depending on the programming model. FlashGraph makes use of selective vertex scheduling to skip some vertices during execution. While this can reduce the number of vertices to access, on the opposite side, it slows down the convergence speed. As a result, the impact of slow convergence is larger and FlashGraph runs significantly slower despite the use of selective scheduling.

6.4 Optimizations

Figure 11 displays the effect of using optimization techniques. The performance numbers were measured using PageRank with multiple optimization combinations. The white bars represent the performance of ExtraV where the worker modules lack stream buffers. The light-gray bars represent the performance when the stream buffers are added to the pipeline but the prefetching is not used, and the dark-gray bars represent the performance with both the stream buffers and prefetching. Without stream buffers, workers rely on the PSL cache to access data outside the FPGA.

Even though the PSL cache has a 256KB capacity, sharing it among multiple worker modules incurs frequent conflicts, resulting in an inefficient use of the cache. Moreover, the latency of the PSL cache from the AFU is around 120 ns [3], which translates to 15 cycles at 125 MHz every time a worker module reads a word. As a result, ExtraV's accelerator without stream buffers suffers almost a 10× slowdown.

To hide the latency of storage devices, software-based frameworks usually rely on highly parallel I/O requests, either by using custom file systems [53] or multiple worker threads [41]. ExtraV employs 16 worker modules, but it's not enough to draw the full bandwidth from the storage without prefetching techniques. With the prefetching, the performance rises up to almost three times.

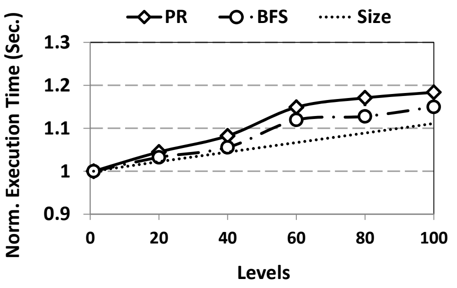

Figure 12: Multi-version traversal performance.

6.5 Multi-version Performance

To see the scalability of multi-version traversal performance, we divided Friendster dataset into 100 versions. We took 90% of the edges at the base version, and the rest of the edges were randomly distributed into the remaining versions. Figure 12 shows the performance trend.

The results show about 0.18% latency increase per version for PageRank and 0.15% for BFS, indicating some overhead in addition to the size of added edges. The overhead from the increasing the number of versions comes from flushing the stream buffers. When traversing a single-versioned graph, the edges in a single array are accessed in order even though there may be jumps between accesses due to partial traversals. On a multi-versioned graph, accesses to edges in each version would still be sequential, but as a whole it would exhibit a random access pattern. It also explains the difference between BFS and PageRank. BFS uses partial traversal more often, and slowdowns from the stream buffer flushes are already happening at version 0. One could mitigate this by having a separate stream buffer for each version, but it would require significant more resources and problem is still there if the number of versions exceed that of the stream buffers. We plan to address this issue in future work.

6.6 Compression Ratio

The compression ratio is the most critical factor that determines the speedup of ExtraV. Table 6 shows the compression ratios of various graphs. To show that our scheme works on a broad range of graphs, we include a few additional datasets. The uncompressed size is measured in the CSR format, and the compressed size includes both the vertex table and the compressed edge list. Graphs from the web usually show high compression ratios, arabic-2002 being the best with 9.36×. This comes from the fact that the vertices from web graphs often have continuously labeled vertices as their neighbors. The Gsh-tpd graph shows a smaller compression ratio than other web graphs. The reason is that the edges of Gsh-tpd graph are not densely packed together since it is not limited to a certain subdomain and covers a much wider area. The differences of vertex indices could be larger and there is less chance of having intervals or small differences within neighbor lists. Social network and citations graphs, on the other hand, show even lower compression ratios. They show less regularity, and the edges span over an even wider range. Livejournal shows the best compression ratio among them with 2.70×, while Twitter, Friendster and MS-ref shows a

Graph	Type	#Vertices	#Edges	Uncompressed	Compressed	Ratio
Gsh-tpd [14]	Web crawl	30,809,122	602,119,716	9.89GB	2.47GB	4.00×
uk-2005 [14]	Subdomain Web	39,459,925	936,364,282	15.8GB	1.71GB	9.24×
arabic-2002 [14]	Subdomain Web	22,744,282	639,999,458	10.2GB	1.09GB	9.36×
Twitter [36]	Social Network	41,652,230	1,468,365,182	23.7GB	9.8GB	2.41×
Friendster [50]	Social Network	124,836,180	1,806,067,135	30.6GB	13.8GB	2.22×
Livejournal [8]	Social Network	4,847,571	68,993,773	1.23GB	456MB	2.70×
MS-ref [7]	Citations	46,742,304	528,682,289	9.27GB	4.40GB	2.10×
Road-TX [8]	Road Network	1,379,917	1,921,660	102MB	54MB	1.89×
Road-CA [8]	Road Network	1,965,206	2,766,607	145MB	76MB	1.90×

Table 6: Compression Ratio of Various Graph Datasets

little over 2×. Vertices in the road networks are physically limited in 2D spaces and have small degrees ranging from 3 to 5. Thus there is relatively small room to compress and the ratio is around 1.9× for both networks.

7. CONCLUSION

In this paper, we propose the ExtraV system that uses a coherent hardware accelerator to gain significant speedup with near-storage processing. Our ExtraV significantly improves the performance of out-of-memory graph processing, thereby closing the gap towards in-memory graph processing. Common graph processing functions are implemented and executed on the AFU (Accelerator Functional Unit) in front of the storage device, while more application specific tasks are implemented as software that runs on the host processor for flexibility. The key technique in ExtraV is the graph virtualization programming model. Using streaming queries, it provides the program running on the host processor with an abstraction that the entire graph dataset resides in memory. To mitigate the fundamental limit of storage bandwidth, an expand-and-filter scheme is applied to graph traversal, which processes the graph data, filters them, and feeds only desired results back to the host processor. Multi-version traversal is also processed by the AFU to provide access to past states and easy modifications.

Our ExtraV system is prototyped on a CAPI-enabled POWER8 processor with a Xilinx Ultrascale FPGA accelerator card. The accelerator functions are designed and implemented using a high-level synthesis flow. System performance results on multiple graph algorithms and datasets indicate that the ExtraV system gains significant speedup compared to software based state-of-the art implementations.

8. REFERENCES

[1] Adm-pcie-ku3. http://www.alpha-data.com/dcp/products.php?product=adm-pcie-ku3.

[2] Capi-streaming-framework. https://github.com/mbrobbel/capi-streaming-framework.

[3] Coherent accelerator processor interface frequently asked questions. https://www.ibm.com/developerworks/community/files/basic/anonymous/api/library/88b7bb80-c4f6-4f85-86af-92b8a44af584/document/c7d4e29d-cff7-45c8-a27c-370a1e07c361/media.

[4] The data explosion in 2014 minute by minute - infographic. http://aci.info/2014/07/12/the-data-explosion-in-2014-minute-by-minute-infographic/.

[5] IBM POWER system S824L. http://www-03.ibm.com/systems/power/hardware/s824l/specs.html.

[6] ICCAD 2014 workshop: Heterogeneous computing platforms (HCP). https://iccad.com/sites/2013.iccad.com/files/files/hcp14-final-program.pdf.

[7] Microsoft academic graph. https://academicgraph.blob.core.windows.net/graph/index.html.

[8] Stanford large network dataset collection. https://snap.stanford.edu/data/index.html.

[9] Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.

[10] Vivado High-Level Synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[11] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*, pages 105–117, 2015.

[12] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *ISCA*, pages 336–348. IEEE, 2015.

[13] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. In *SC*, pages 1–10, 2012.

[14] P. Boldi, A. Marino, M. Santini, and S. Vigna. BUbiNG: Massive crawling for the masses. In *WWW*, pages 227–228, 2014.

[15] P. Boldi and S. Vigna. The Webgraph framework I: Compression techniques. In *WWW*, pages 595–602, 2004.

[16] M. H. Brad Brech, Juan Rubio. IBM data engine for NoSQL. *IBM White Paper*, 2014.

[17] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[18] S. Brin and L. Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer networks*, 56(18):3825–3833, 2012.

[19] M. Ceriani, S. Secchi, O. Villa, A. Tumeo, and G. Palermo. Exploring efficient hardware support for applications with irregular memory patterns on multinode manycore architectures. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2014.

[20] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD*, pages 219–228, 2009.

[21] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[22] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart SSDs: Opportunities and challenges. In *SIGMOD*, pages 1221–1230, 2013.

[23] H. Giefers, R. Polig, and C. Hagleitner. Accelerating arithmetic kernels with coherent attached FPGA coprocessors. In *DATE*, pages 1072–1077, 2015.

[24] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[25] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. Biscuit: A framework for near-data processing of big data workloads. In *ISCA*, ISCA '16, pages 153–165, Piscataway, NJ, USA, 2016. IEEE Press.

[26] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman. AC-DIMM: Associative computing with STT-MRAM. In *ISCA*, pages 189–200, New York, NY, USA, 2013. ACM.

[27] P. K. Gupta. Xeon+ FPGA platform for the data center. In *CARL*, 2015.

[28] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO*, pages 1–13, 2016.

[29] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD*, pages 77–85, 2013.

[30] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC*, pages 197–208. 2007.

[31] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi. PGX.D: A fast distributed graph processing engine. In *SC*, pages 58:1–58:12, 2015.

[32] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *PACT*, pages 78–88, 2011.

[33] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. Simplifying scalable graph processing with a domain-specific language. In *CGO*, pages 208:208–208:218, New York, NY, USA, 2014. ACM.

[34] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. Bluedbm: An appliance for big data analytics. In *ISCA*, pages 1–13, New York, NY, USA, 2015. ACM.

[35] U. Kang, C. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system implementation and observations. In *ICDM*, pages 229–238, 2009.

[36] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[37] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[38] J. Lee, J. H. Ahn, and K. Choi. Buffered compares: Excavating the hidden parallelism inside DRAM architectures with lightweight logic. In *DATE*, pages 1243–1248, March 2016.

[39] J. Lee, J. Chung, J. H. Ahn, and K. Choi. Excavating the hidden parallelism inside dram architectures with buffered compares. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Preprint, 2017.

[40] 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. *VLDB*, 5(8):716–727, Apr. 2012.

[41] P. Macko, V. Marathe, D. Margo, and M. Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *ICDE*, pages 363–374, 2015.

[42] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[43] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki. PrefEdge: SSD prefetcher for large-scale graph traversal. In *SYSTOR*, pages 4:1–4:12, 2014.

[44] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. Energy efficient architecture for graph analytics accelerators. In *ISCA*, pages 166–177, 2016.

[45] B. Przybus. Xilinx redefines power, performance, and design productivity with three new 28 nm FPGA families: Virtex-7, Kintex-7, and Artix-7 devices. *Xilinx White Paper*, 2010.

[46] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[47] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: a user-programmable ssd. In *OSDI*, pages 67–80, 2014.

[48] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, Jan 2015.

[49] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[50] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *ICDM*, pages 745–754, 2012.

[51] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee. Fast iterative graph computation: A path centric approach. In *SC*, pages 401–412, 2014.

[52] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: Throughput-oriented programmable processing in memory. In *HPDC*, pages 85–98, 2014.

[53] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *FAST*, pages 45–58, 2015.

[54] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical Report CMU-CALD-02-107, Carnegie Mellon University, 2002.

[55] X. Zhu, W. Han, and W. Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, pages 375–386, 2015.