GraFBoost: Using accelerated flash storage for external graph analytics

Sang-Woo Jun[†], Andy Wright[*], Sizhuo Zhang[*], Shuotao Xu[†] and Arvind[†]

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
[†]{*wjun,shuotao,arvind*}*@csail.mit.edu* [*]{*acwright,szzhang*}*@mit.edu*

***Abstract*—We describe GraFBoost, a flash-based architecture with hardware acceleration for external analytics of multi-terabyte graphs. We compare the performance of GraFBoost with 1 GB of DRAM against various state-of-the-art graph analytics software including FlashGraph, running on a 32-thread Xeon server with 128 GB of DRAM. We demonstrate that despite the relatively small amount of DRAM, GraFBoost achieves high performance with very large graphs no other system can handle, and rivals the performance of the fastest software platforms on sizes of graphs that existing platforms can handle. Unlike in-memory and semi-external systems, GraFBoost uses a constant amount of memory for all problems, and its performance decreases very slowly as graph sizes increase, allowing GraFBoost to scale to much larger problems than possible with existing systems while using much less resources on a single-node system.**

The key component of GraFBoost is the *sort-reduce* accelerator, which implements a novel method to sequentialize fine-grained random accesses to flash storage. The sort-reduce accelerator logs random update requests and then uses hardware-accelerated external sorting with interleaved reduction functions. GraFBoost also stores newly updated vertex values generated in each *superstep* of the algorithm lazily with the old vertex values to further reduce I/O traffic.

We evaluate the performance of GraFBoost for PageRank, breadth-first search and betweenness centrality on our FPGA-based prototype (Xilinx VC707 with 1 GB DRAM and 1 TB flash) and compare it to other graph processing systems including a pure software implementation of GrapFBoost.

***Keywords*-graph analytics; flash storage; FPGA; hardware acceleration; sort-reduce**

I. Introduction

Extremely large and sparse graphs with irregular structures (billions of vertices and hundreds of billions of edges) arise in many important problems, for example, analyses of social networks [1], [2], and structure of neurons in the brain [3]. Their efficient processing enables everything from optimal resource management in power grids [4] to terrorist network detection [5].

Previous competitive systems for solving large-scale graph analytics problems have taken one of two approaches: (1) provision the system with enough DRAM to store the entire graph data structure, then perform random accesses to the graph directly out of DRAM, or (2) provision the system with enough DRAM to store only the vertex data in DRAM, then stream the edges in from secondary storage (disk or flash). Both approaches require a substantial amount of DRAM as they aspire to solve larger graph problems, thereby forcing users to buy increasingly expensive systems [6], [7], [8]. Today it is routine, for example, to use multiple servers to solve large graph problems to obtain the amount of DRAM required to store the relevant graph data structures [9], [10].

A. GraFBoost

We present GraFBoost, a flash-storage system for large-scale graph analytics computations providing a vertex-centric programming model. GraFBoost implements a novel *sort-reduce* algorithm (Section III), which sequentializes and coalesces fine-grained (otherwise random) accesses to the underlying graph data structure. This algorithm enables GraFBoost to convert expensive random accesses into efficient streamed block SSD accesses. As a result, GraFBoost can efficiently execute large-scale graph analytics computations (we present experimental results for graphs of up to 4 billion vertices and up to 128 billion edges) efficiently on single-node devices with bounded DRAM (1 GB in our implemented system). Moreover, our approach scales independently of the amount of available DRAM, instead depending only on the size of storage required. We report results for two implementations of GraFBoost:

- **Software Only (GraFSoft) :** Executed on a standard 32 core (two sockets) Xeon server with up to 128 GB DRAM (of which the implementation uses 16 GB) and 2.5 TB of attached flash storage.
- **Hardware Accelerator (GraFBoost):** Uses a hardware accelerator, implemented in a VC707 FPGA board with 1 TB of attached flash storage. The accelerator is attached to the same server as the software only implementation, but uses very limited hardware resources, e.g., 2 GB of DRAM, and two threads.

We compare the performance of our two single-node GraFBoost implementations with single-node implementations of other graph analytics systems, using algorithms implemented by the same developers. Specifically, we compare GraFBoost to (1) GraphLab [11], (2) FlashGraph [7], (3) X-stream [12], and (4) GraphChi [13], using the same environment as GraFSoft. We evaluated these systems on the twitter graph, the web data commons graph [14] with

3 billion vertices and 128 billion edges, and three graphs synthesized according to the Graph500 [15] requirements, ranging from a 250 million to 4 billion vertices.

B. Comparative Results Summary

GraphLab stores the entire graph, both vertices and edges, in DRAM. When the graph fits in DRAM, GraphLab exhibits the best performance over all systems except for X-Stream on PageRank (see Section V). When the graph does not fit in DRAM (this is the case for 4 of our 5 benchmark graphs), GraphLab thrashes swap space and fails to complete within reasonable time.

FlashGraph stores all vertices in DRAM. When the vertices fit, FlashGraph exhibits performance comparable to GraphLab. Because it stores edges in flash, FlashGraph can work successfully with larger graphs (4 out of our 5 benchmark graphs). For these graphs, FlashGraph exhibits performance comparable to our hardware accelerator performance, better than X-stream and GraphChi, and between two to four times faster than our software only implementation. However, for graphs that are even larger (1 out of our 5 benchmark graphs), even the vertex data fails to fit in DRAM and FlashGraph fails to complete.

X-stream is designed to work with large graphs and little DRAM. The X-stream algorithm traverses the entire graph on every superstep in the graph analytics computation. X-stream runs slower than GraFBoost on all but one of the benchmarks and exhibits extremely poor performance when the computation requires many, sparse supersteps (as is the case for one of our benchmarks).

GraphChi is designed to work entirely out of secondary storage (specifically disk). Its performance is not competitive with any of the other systems, including GraFBoost.

GraFBoost (both software only and hardware accelerator versions) processes all of our five benchmarks successfully, with all benchmarks completing in less than 30 minutes. No other system performed all of our benchmark graph computations successfully. Even for smaller graphs for which other systems could complete successfully, GraFBoost demonstrated performance comparable to the fastest systems evaluated. These results highlight the ability of GraFBoost to scale to large graph computations independently of the amount of DRAM and with predictable performance.

C. Hardware Accelerator

The GraFBoost hardware accelerator implements the sort-reduce algorithm. In this capacity, it can offer substantial performance gains (typically between a factor of two to four) over the software only solution implemented on a standard microprocessor. Our current implementation uses a single VC707 FPGA development board. We note that SSD flash controller chips typically contain around 4 to 8 ARM cores and 500 MB to 1 GB (or more) of DRAM [16]. Since our accelerator uses less than half of the FPGA on a single VC707 development board, an ASIC version of our accelerator could reasonably fit on an SSD flash controller.

The GraFBoost hardware accelerator provides high enough performance to saturate the bandwidth of the flash storage and available DRAM, reducing the overhead of sequentializing access. Furthermore, it offloads most of the computation from the CPU, freeing it to do other useful work.

D. Contributions

- The novel sort-reduce method of vertex updates to achieve high performance graph analytics on flash storage, including the important optimization of interleaving sorting and reduction operations;
- Design and implementation of a hardware accelerated flash storage device and its flexible programming environment to implement large graph algorithms;
- Effective data structures and algorithms for graph analytics that focus on reducing flash I/O and memory usage; and
- An in-depth analysis of the performance impact of our innovations on a variety of system configurations including a multithread software implementation of hardware accelerators.

The rest of this paper is organized as follows: Section II introduces existing work on large graph analytics, flash storage and hardware acceleration. Section III describes in detail our newly proposed sort-reduce algorithm and how it can be used for graph analytics. Section IV explains the internal architecture of our hardware and software designs, and Section V presents their performance and resource utilization evaluation. We conclude with future work in Section VI.

II. Related Work

A. Large Graph Analytics

The irregular nature of graphs makes graph analytics fine-grained random access intensive, and typically requires all data to fit in DRAM [17], [9]. However, as graphs become larger, they quickly become too large to fit in the main memory of a reasonable machine, and must be distributed across a cluster [6]. Large-scale graph analytics are usually done using a distributed graph processing platform following a programming model it provides, so the user does not have to deal with the difficulties of distribution, parallelism and resource management [18], [19], [20], [21].

Many prominent graph analytics platforms, including Pregel [22], Giraph [23] and GraphLab [11], expose a vertex-centric programming model [24] because of its ease for distributed execution. In a vertex-centric model, a graph algorithm is deconstructed so that it can be represented by running a *vertex program* on each of the vertices. A vertex program takes as input information about the current vertex, as well as its neighboring vertices and the edges that connect them. After execution, a vertex program updates the current vertex, and possibly sends messages to neighboring vertices. Vertex-centric systems can further be categorized into two paradigms; Pull-style systems, in which the program reads the values of neighboring vertices and updates its own value, and Push-style systems, in which the program updates the values of its neighbors [25]. In the push-style system, each vertex's value is updated as many times as its incoming neighbors, whereas in pull-style systems updates happen once for each vertex. Some systems, like Giraph++ [26] and Blogel [27] expand the vertex-centric model into a subgraph-centric model, in which systems process blocks of subgraphs and messages between them.

On the other hand, some systems such as X-Stream [12] and GridGraph [28] provide edge-centric programming models, which is aimed at sequentializing accesses into edge data stored in secondary storage, which is usually significantly larger than vertex data. Edge-centric systems have the benefits of doing completely sequential accesses to the edge data, but must processes all edges in the graph at every superstep, making them inefficient for algorithms with sparse active lists like breadth-first search.

Some recent systems such as Combinatorial BLAS [29], GraphBLAS [30], and Graphulo [31] provide the user a set of linear algebra functions designed for graph algorithms [32]. While both the vertex-centric and linear-algebraic models are versatile enough for many important applications, some applications, such as accurate triangle counting and community detection still require finer grained control of execution. Galois [25] is one such framework and it has high-performance platform implementations both on shared and distributed memory systems.

Not only do these platforms differ on how the algorithm is expressed, they also differ on how the algorithms are executed. Many graph analytics systems execute vertex programs in supersteps in a *bulk synchronous* manner, in disjoint *supersteps* [22], [33], [34]. A vertex program in a superstep can only read from the previous superstep. Some other systems provide an asynchronous environment, where each vertex program sees the latest value from neighboring vertices, and can execute asynchronously [11], [35]. While asynchronous systems are more complex compared to bulk synchronous systems, sometimes it results in faster convergence for many machine learning problems [11].

Semi-external systems achieve high performance with less memory by storing only the vertex data in memory, and optimizing access to the graph data stored in secondary storage. Since vertex data is much smaller than edge data, semi-external systems need much smaller DRAM capacity than in-memory systems. Semi-external systems such as FlashGraph [7] can often achieve performance comparable to completely in-memory systems as long as the vertex data can completely fit in memory. X-Stream [12] also achieves high performance by keeping vertex data in memory. It logs updates to vertex values before applying them, so when available memory capacity is low they can easily partition operations without losing much performance, by simply splitting the stream. This kind of partitioning is not readily applicable to vertex-centric systems such as FlashGraph [7], because reading the value of neighboring vertices for each vertex program execution requires fine-grained random access.

When graph sizes become too large even to store only vertex data in DRAM, external systems like GraphChi [13] become the only choice due to their lower memory requirements. GraphChi re-organizes the algorithm to make data access completely sequential, and thus, make accesses perfectly suitable for coarse-grained disk access. There is also active research on optimizing storage access for such external graph analytics systems [36]. However, GraphChi does so by introducing additional work, and requires the whole graph data to be read multiple times each iteration. These extra calculations result in low performance on large graphs and makes GraphChi uncompetitive with memory-based systems. Some systems including LLAMA [37] and MMap [38] use OS-provided `mmap` capabilities to make intelligent on-request storage access.

On the other hand, some systems, including Ligra [39], aim to optimize performance for data that can fit in the memory of a single machine, by making the best use of shared memory parallel processing.

Another way to deal with the large amount of graph data is to use a graph-optimized database, such as neo4j [40], which provides an ACID-compliant transactional interface to a graph database. There has been efforts to use generic RDBM engines for graph analytics instead of specialized systems [41].

B. Flash Storage

Many researchers have explored the use of secondary storage devices such as flash storage to counter the cost of large clusters with lots of DRAM, overcoming many of the challenges of using secondary storage for computation. While the slow performance and long seek latency of disks prohibited serious use of external algorithms, flash storage is orders of magnitude faster both in terms of bandwidth and latency [42]. However, flash storage still poses serious challenges in developing external algorithms, including still a much lower bandwidth compared to DRAM, higher latency in the range of 100s of microseconds compared to tens of nanoseconds of DRAM, and a coarser access granularity of 4KB to 8KB *pages*, compared to DRAM's cache line [42]. Large access granularity can create a serious storage bandwidth problem for fine-grained random access; if only 4 bytes of data from an 8KB flash page is used, the bandwidth is reduced effectively by a factor of 2048. As a result, naive use of secondary storage as a memory extension will result in a sharp performance drop, to the point where the system is no longer viable.

Flash also suffers from lifetime issues due to erase and write causing physical wear in its storage cells. Commercial flash storage devices handle wearing using an intermediate software called the Flash Translation Layer (FTL), which tries to assign writes to lesser-used cells, in a technique called *wear leveling* [43]. Flash cell management is made even more difficult by the requirement that before pages can be written to, it must be *erased*, an expensive operation of even coarser *block* granularity of megabytes. Random updates handled naively will cause a block erase every time, requiring all non-updated data to be read and re-written to the erased block. Modern systems try to avoid this using many techniques including log-structured file systems [44], [45], but random updates are often still expensive operations [42]. There is also active work in file systems that manage flash chips directly in order to remove the latency overhead of FTLs [46], [47], [48], as well as optimized networks for flash arrays [49], [50].

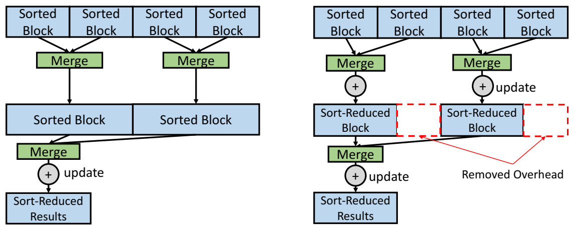
C. Special Hardware for Graph Analytics

As performance scaling of CPUs slow down [51], many systems have attempted to efficiently use newer computation fabric to accelerate analytics, such as domain-specific architectures for database queries [52], [53], [54], [55], in-memory graph specific processors such as GraphCREST [56], Graphicionado [57] and other processor architectures optimized for graph processing [58], [59], [60], [61]. GraphGen [62] attempts to demonstrate high performance and power-efficient in-memory graph analytics using FPGAs by automatically compiling an application-specific graph processor and memory system. Coupled with good FPGA hardware, GraphGen was able to outperform handwritten CPU and GPU programs sometimes by over 10x. ForeGraph [63] uses a cluster of FPGAs to accommodate a graph in its collective on-chip memory. Platforms such as GunRock [64] organize graph processing to be effectively parallelized in a GPU. However, all such systems are limited by their on-chip memory or DRAM capacity in their ability to handle large graphs.

As data sizes become large and more resident in secondary storage, there has been a lot of work in near-storage processing, where computation is moved to the storage to reduce data movement overhead [65], [66], [67], [68], [69], [70], even using reconfigurable accelerators [71], [72] or GPUs [73]. Similar near-data efforts are being made for DRAM as well [74], [75].

III. Sort-Reduce Accelerator for Efficient Vertex Updates

The key component of GraFBoost is a Sort-Reduce accelerator that sequentializes a stream of random updates to an array stored in secondary storage.



(a) Completely sorting before applying update

(b) Interleaving sorting and update

Figure 1: Interleaving sorting and updates dramatically reduces the amount of partially sorted data to be sorted

A. The Sort-Reduce Method

Suppose we want to update an array x and we are given a list xs of key-value pairs $\langle k,v\rangle$ and a function f as follows:

```
for each $\langle k,v\rangle$ in $xs$ do
    $\mathbf{x}[k] = f(\mathbf{x}[k], v)$
end for
```

For example, to compute the histogram of list xs, f would be the addition operator, and x would be initialized to 0.

Consider performing this computation when both array x and list xs are large, i.e., contain hundreds of millions to billions of elements, and the distribution of keys in xs is extremely irregular with many duplicate keys. This computation would result in a lot of fine-grained random updates to the elements of x, rendering caching ineffective. If x was stored on disk then these fine-grained random access will also result in extremely poor use of the disk bandwidth.

One way to make more efficient use of storage bandwidth could be to sort the list xs by key before applying the reduction function f. Once the list is sorted, updates to x, instead of being random, will become completely sequential.

Another way to make more efficient use of storage bandwidth is to merge entries in xs with matching keys as long as the function f is binary associative, that is, $f(f(v_1,v_2),v_3) = f(v_1, f(v_2,v_3))$. For example the entires $\langle k,v_1\rangle$ and $\langle k,v_2\rangle$ can be merged into a single entry $\langle k, f(v_1,v_2)\rangle$. This reduces the size of xs, and therefore reduces the amount of key-value pairs accessed when updating x.

These two methods are leveraged together in the *sort-reduce* method (see Figure 1). In this method, merge steps in the sorting algorithm are interleaved with reduction operations to merge key-value pairs for matching keys to reduce the size of xs at each sorting step. After each merge step, if two consecutive values have matching keys, they are merged into a single key-value pair before performing the next merge step.

B. Sort-Reduce Accelerator

We have implemented a hardware sort-reduce accelerator which offloads computation intensive sorting operations from the CPU. The accelerator is parameterized by the

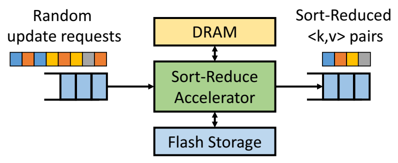


Figure 2: Dataflow involved with the sort-reduce accelerator

reduction operator f and has the following signature (Figure 2):

$$SortReduce_f :: unsorted\ list\ of\ \langle k,v\rangle\ pairs$$
$$\rightarrow list\ of\ sorted\ and\ reduced\ \langle k,v\rangle\ pairs$$

The detailed architecture of the sort-reduce accelerator is described in Section IV. There are several important points to be noted about our design: First, if the size of input list (*xs*) is small then it is completely sorted in memory. Thus, the decision about when to use internal DRAM based sorting and when to use external sorting is internal to the accelerator and is based on the data set size and the available amount of DRAM. Second, the accelerator can use either a sparsely or densely encoded representation for the output list.

We have also implemented a software version of the sort-reduce accelerator for verification and performance-comparison purposes. Of course the software version requires a server class machine to show similar performance as will be discussed in Section IV-F

C. Using Sort-Reduce for External Graph Analytics

Many graph algorithms are expressed in the push-style vertex-centric programming model shown in Algorithm 1. The programmer provides definitions for algorithm-specific functions *edge_program*, *vertex_update*, *finalize*, and *is_active*. In each *superstep* i, one goes through the current *active vertex* list A_{i-1}, and computes the new temporary vertex values $newV_i$ using the *edge_program* and *vertex_update* functions. We then compute the new vertex values by applying the *finalize* function to each element of $newV_i$. In many algorithms including breadth-first search and single-source shortest path, the vertices in the new active list are a subset of the vertices in $newV_i$. For such algorithms, one can determine whether a vertex is active or not by comparing against its value from the previous superstep V using the function *is_active*. The values of the newly computed vertex list A_i is applied to the current vertex value array V. The process is repeated for certain number of supersteps or until the active list becomes empty.

Computing $newV_i$ faces the same fine-grained random update problem in that we discussed in Section III-A, because the distribution of k_{dst} is irregular. We can reformulate Algorithm 1 using the sort-reduce method to get Algorithm 2. Algorithm 2 first logs all the updates in list U and feeds it to the SortReduce accelerator to produce $newV_i$. We later show that applying sort-reduce for graph analytics has dramatic performance benefits, and allows GraFBoost to compete with other systems with much larger DRAM capacity.

The associativity restriction for *vertex_update* required by sort-reduce is not very limiting, and it can represent a large amount of important algorithms. It is also a restriction shared with many existing models and platforms, including the Gather-Apply-Scatter model used by PowerGraph [35] and Mosaic [76], as well as the linear algebraic model.

The vertex value V is stored as an array of vertex values in storage, and is accessed using the key as index. Because $newV_i$ and A_i are all sorted by key, access to V is also always in order. As a result, the performance impact of accessing and updating V is relatively low.

We can further reduce storage I/O by evaluating A_i element-by-element lazily only when it is required by the next superstep. Instead of immediately computing and storing A_i from one iteration to the next, we can iterate over $newV_{i-1}$ at the next superstep and determine on the fly if the accessed vertex k_{src} is active or not. If the vertex k_{src} is active then we can apply the *edge_program* to its outgoing edges as before and also update $V[k_{src}]$ at the same time. The transformed Algorithm is shown in Algorithm 3 and it does two fewer I/O operations per active vertex.

In some algorithms where the active list is not a subset of $newV$, the programmer can create a custom function that generates the active vertex list. For example, the active list for the PageRank formulation in Algorithm 4 is the set of vertices which are sources of edges that point to a vertex in $newV$. In this implementation, the next iteration will perform PageRank on the active vertices to update the PageRank of the values in $newV$.

In this implementation a bloom filter is used to keep track of all vertices which have edges that point to a vertex in $newV$. $V[k]$ also stores the superstep index i when it was updated, so that sort-reduced values for vertices that were not included in the previous superstep's $newV$ can be ignored. In the interest of performance, bloom filter management and looping through the keyspace can also be implemented inside the accelerator.

IV. System Architecture

Figure 5 shows the overall architecture of a GraFBoost system. A GraFBoost storage device, which consists of NAND flash storage and a FPGA-based in-storage hardware accelerator, is plugged into a host machine over PCIe, which can be a server, PC or even a laptop-class computer. The GraFBoost storage device uses a custom-designed flash storage hardware [72], [77] to expose raw read, write and erase interfaces of its flash chips to the accelerator and host, instead of through a Flash Translation Layer (FTL). All the data structures such as the index and edge lists to describe the graph structure, vertex data V, $newV$, and temporary data structures created and deleted during sort-reduce operation

Algorithm 1 A baseline algorithm for performing a vertex program superstep

```
for each $\langle k_{src}, v_{src} \rangle$ in $A_{i-1}$ do
    ▷ Iterate through outbound edges
    for each $e(k_{src}, k_{dst})$ in G do
        $ev$ = edge_program($v_{src}$, $e.prop$)
        $newV_i[k_{dst}]$ = vertex_update($ev, newV_i[k_{dst}]$)
    end for
end for
for each $\langle k, v \rangle$ in $newV_i$ do
    $x$ = finalize($v, V[k]$)
    if is_active(x,$V[k]$) then
        $A_i$.append($\langle k, x \rangle$)
    end if
end for
$V$ = update($V$, $A_{i-1}$)
```

Algorithm 2 Using sort-reduce for performing a vertex program superstep

```
U = []
for each $\langle k_{src}, v_{src} \rangle$ in $A_{i-1}$ do
    for each $e(k_{src}, k_{dst})$ in G do
        $ev$ = edge_program($v_{src}$, $e.prop$)
        U.append($\langle k_{dst}, ev \rangle$)
    end for
end for
$newV_i = SortReduce_{vertex\_update}(U)$
for each $\langle k, v \rangle$ in $newV_i$ do
    $x$ = finalize($v, V[k]$)
    if is_active($x, V[k]$) then
        $A_i$.append($\langle k, x \rangle$)
    end if
end for
$V$ = update($V$, $A_{i-1}$)
```

Algorithm 3 Using sort-reduce with lazy evaluation of active vertices

```
U = []
for each $\langle k_{src}, v_{src} \rangle$ in $newV_{i-1}$ do
    $x$ = finalize($v_{src}, V[k_{src}]$)
    if is_active($x, V[k_{src}]$) then
        for each $e(k_{src}, k_{dst})$ in G do
            $ev$ = edge_program($v_{src}$, $e.prop$)
            U.append($\langle k_{dst}, ev \rangle$)
        end for
        $V[k_{src}] = x$
    end if
end for
$newV_i = SortReduce_{vertex\_update}(U)$
```

Figure 3: Transforming a vertex program to use the sort-reduce accelerator

Algorithm 4

```
$bloom$ = [], U = []
for each $\langle k, v \rangle$ in $newV_{i-1}$ do
    $x$ = finalize($v, V[k]$)
    if is_active($x, V[k]$) then
        for each edge $e(k_{src}, k)$ in graph G do
            $bloom[k_{src}] = true$
        end for
        $V[k_{src}] = \langle x, i \rangle$
    end if
end for
for all $\langle k_{src}, \_ \rangle$ in graph G do        ▷ Loop through keyspace
    if $bloom[k_{src}] = true$ then
        for each edge $e(k_{src}, k_{dst})$ in graph G do
            $ev$ = edge_program($V[k_{src}], e.prop$)
            U.append($\langle k_{dst}, ev \rangle$)
        end for
    end if
end for
$newV_i = SortReduce_{vertex\_update}(U)$
```

Figure 4: Sort-reduce with custom active list generation for PageRank

are maintained as files and managed by the host system using a lightweight flash file management system called the Append-Only Flash File System (AOFFS).

The host system executes supersteps one at a time. It initiates each superstep by reading *newV*, parts of *V* and the graph files, and feeding the information to the edge program. The stream produced by the edge program is fed into the sort-reduce accelerator, which sorts and applies vertex updates to create a *newV* for the next superstep. We first describe the storage data structures, and then the accelerators involved in graph analytics.

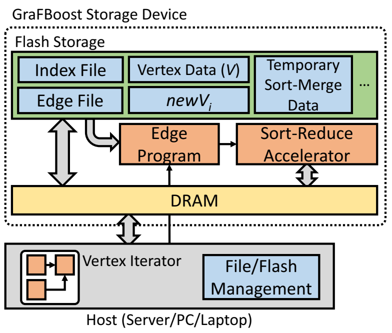


Figure 5: The internal components of a GraFBoost system

A. Append-Only Flash File System

AOFFS manages the logical to physical mapping for flash storage in the host instead of an FTL, much like AMF [48] or Noftl [47]. The major characteristic of AOFFS is that for each file, writes can only happen at the end of the file, by appending data. This is enough for GraFBoost because thanks to sort-reduce, it does not require random updates at any point during execution. By removing the random update requirement, flash management overhead becomes significantly simpler, and drastically reduces the access latency of flash from host.

B. Data Representation

GraFBoost stores graphs in a compressed sparse column format, or outbound edge-list format in graph terminology, as shown in Figure 6. For each graph, we keep an Index and an Edge file in flash storage. For the applications discussed in this paper, these files do not change during execution. In order to access the outbound edges and their destinations from a source vertex, the system must first consult the index file to determine the location of the outbound edge information in the edge file. For some algorithms like PageRank, an inbound edge information is also required, which is stored in the same format. The vertex data file V is stored as a dense array of vertex data. However, the sort-reduce result $newV$, as well as the temporary files created by the sort-reduce accelerators are stored in a sparse representation, which is a list of key-value pairs.

C. Effect of fixed-width access to DRAM and flash

In order to make the best use of DRAM and flash bandwidth at a certain clock speed, the interface to these devices must be wide enough to sustain their bandwidth. For example, the accelerators in our GraFBoost prototype uses 256-bit words for communication. We pack as many elements as possible in these 256-bit words, ignoring byte and word alignment, but without overflowing the 256-bit word.

For example, if the key size is 34 bits, it will use exactly 34 bits instead of being individually padded and aligned to 64 bits (see Figure 7). Such packing and unpacking entails no overhead in specialized hardware and at the same time it saves a significant amount of storage access bandwidth.

D. Edge Program Execution

An edge program function is applied to the value of each active vertex and the edge property of each of its outbound edges. GraFBoost is parameterized by edge program function, which itself is parameterized by vertex value type and edge property type. For single-source shortest-path, the edge program adds the vertex and edge values and produces it as a vertex value. These value types can vary in size from 4 bits to 128 bits. The dataflow involved in edge program execution can be seen in Figure 8.

In order to apply the edge program, GraFBoost must first determine the active vertex list for the current iteration, by streaming in $newV$ and corresponding V elements from storage, and using the programmer-provided *is_active* function. For algorithms that do not fit this model such as PageRank, where the active vertex list is not a subset of $newV$, the programmer can choose to implement custom programs in software that use files in storage and generates a stream

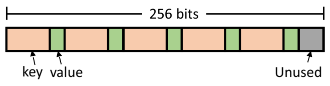


Figure 7: Data packing in a 256-bit word

of active vertex lists. GraFBoost also provides a *vertex list generator* module in hardware for when all nodes are active with default values. It emits a stream of active vertex key-value pairs with uniform values, and the user can provide the range of keys it generates, as well as the value.

After outbound edges of each active vertex is read from the graph files, they can be fed to the edge program. The GraFBoost system operates an array of parallel edge program instances because the wide interface from flash storage may have many values encoded in it. The resulting values from the edge programs are coupled with the destination vertex index of each edges to create a stream of 256-bit wide intermediate key-value pairs. Before this stream is emitted, it is packed so that each emitted 256-bit word has as many values in it as possible.

E. Hardware Implementation of Sort-Reduce

The sort-reduce accelerator operates on the unsorted stream of $\langle k,v\rangle$ pairs coming out the edge program to create the sorted and reduced $\langle k,v\rangle$ list of newly updated vertex values. The design of sort-reduce accelerator is based on the design of a high-performance hardware-accelerated external sorter [78]. We need to first describe the architecture of the external sorter before explaining the architecture of sort-reduce accelerator.

1) Hardware-Accelerated External Sorting: There are three different types of memories of various capacities and speed available for sorting and we exploit the characteristics of each to the fullest extent. These are the external flash storage, 1 GB to 2 GB DRAM on the FPGA board and 4 MB of BRAM inside the FPGA where all the accelerators are implemented. We describe the whole sorting process top down.

We first read an unsorted block of, say 512 MB, from the external storage and sort it in-memory using the DRAM on the FPGA board. We can instantiate as many in-memory sorters as needed to match the flash storage bandwidth,

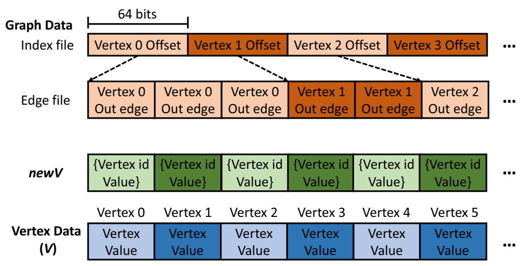


Figure 6: Compressed column representation of graph structure

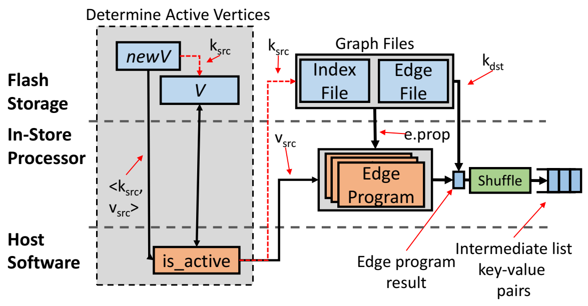
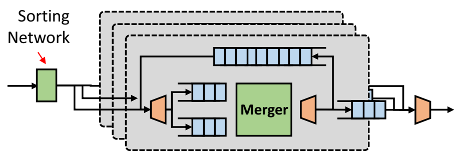
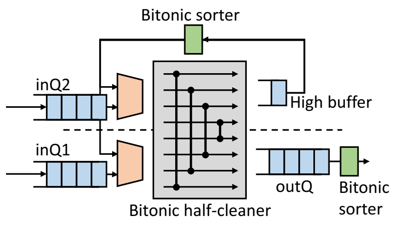


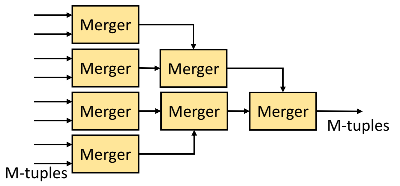
Figure 8: Intermediate list generation



(a) An on-chip sorter makes passes over a chunk of tuples in on-chip memory until sorted

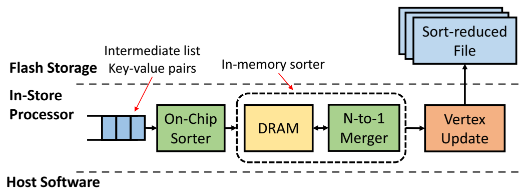


(b) A 4-tuple bitonic merger merges two streams of sorted 4-tuples into a single sorted stream

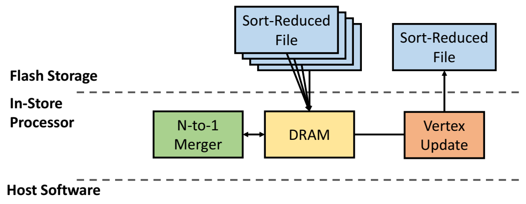


(c) An 8-to-1 merger uses a tree of tuple mergers to merge 8 sorted streams into one

Figure 9: Accelerators components involved in sort-reduce



(a) Intermediate list is sorted first in on-chip memory granularities and then in DRAM



(b) External sorting merges sorted chunks in storage into a new sorted chunk in storage

Figure 10: Sort-reduce accelerator implements various sorters for each stage in the memory hierarchy

provided they fit in the FPGA. The size of the sorting block depends upon the amount of available DRAM and the number of sorters.

We then merge, say 16, sorted blocks by simultaneously reading each from the flash storage to produce a sorted block of size 16×512 MB and write it back in the flash storage. This merger is done in a streaming manner using a 16-to-1 merger and DRAM to buffer input and output. This process is repeated until the full dataset has been sorted.

Each level of sorting is performed using the same strategy: data is streamed in either from on-FPGA memory, DRAM, or flash storage, and then it is sorted using a merge sort network and streamed back out. Each stream of key-value data is split into 256-bit tuples containing multiple key-value pairs.

There are three variants of sorting networks used for different levels of sorting. The first sorting phase uses an on-chip sorter to initially sort 8KB pages of tuples by first sorting each individual tuple with a small sorting network, and then making multiple passes through a merge sort network (see Figure 9a). The later sorting phases use larger merge trees, typically 16-to-1, similar to the 8-to-1 merge tree seen in Figure 9c. Each merger in this tree is a 4-tuple bitonic merger as seen in Figure 9b.

Now we will discuss how this type of external sorting scheme can be turned into a sort-reduce accelerator.

2) Introducing the reduction operation: In order to perform sort-reduce on the intermediate list generated by the edge program, we essentially implement external sort exactly as described above with two salient differences. First, instead of starting with an unsorted list stored in flash storage, it is streamed from the edge program directly to the in-memory sorter, which will sort chunks of 512MB blocks and store them in flash storage. Second, we include the reduction operation before each flash write, implemented as a vertex update function. The dataflow for in-memory sort can be seen in Figure 10a, and the dataflow for external merging can be seen in Figure 10b.

F. Software Implementation of Sort-Reduce

We have also implemented a software version of sort-reduce, both to validate our hardware implementation and to evaluate the benefit of the sort-reduce method by doing a fair comparison with other software systems using the same hardware environment. The software sort-reducer maintains a pool of in-memory sorter threads that perform in-memory sort-reduce on 512MB chunks of memory, and then writes them as individual files. Sorted files are repeatedly merged with a software 16-to-1 merge-reducer until all files have been merged. The performance of a single 16-to-1 is important because eventually all chunks need to be merged into one by a single merger. Due to the performance limitation of a single thread, a multithreaded 16-to-1 merger is implemented as a tree of 2-to-1 mergers, each running on a separate thread. Figure 11 shows the structure of a software merge-reducer. Sorted streams are transferred in large 4 MB chunks in order to minimize inter-process communication overhead. Unlike the hardware implementation, keys and values are word aligned in memory and not densely packed in 256-bit tuples.

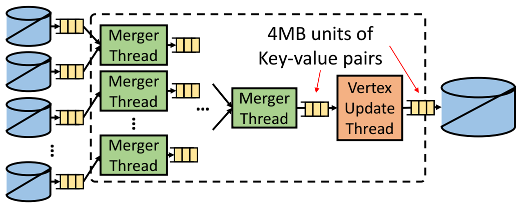


Figure 11: A multithreaded software implementation of sort-reduce

V. Evaluation

We will show the GraFBoost system is desirable for the following two reasons:

1) Using the sort-reduce method in software implementations of large graph analytics (GraFSoft) provides better performance than other in-memory, semi-external, and external systems. By large graphs we mean that the memory requirements for in-memory and semi-external systems exceed the available memory of a single node. We will also show that given a fixed amount of memory, GraFSoft can efficiently process much larger graphs than other systems.
2) The sort-reduce method benefits from hardware acceleration (GraFBoost) both by increased performance and decreased energy consumption. As a result, GraFBoost can not only handle graphs much larger than any other system tested, it provides comparable performance even for smaller graphs which other systems are optimized for.

We support these results through evaluations of multiple graph algorithms and graph sizes using various graph analytics frameworks and our own software and hardware implementations of GraFBoost.

A. Graph Algorithms

The following three algorithms were chosen for evaluation because their implementations were were provided by all platforms evaluated. By using the implementations optimized by the platform developers, we are able to do fair performance comparisons.

Breadth-First-Search: BFS is a good example of an algorithm with sparse active vertices. BFS is a very important algorithm because it forms the basis and shares the characteristics of many other algorithms such as Single-Source Shortest Path (SSSP) and Label Propagation.

BFS maintains a parent node for each visited vertex, so that each vertex can be traced back to the root vertex. This algorithm can be expressed using the following vertex program. *vertexID* is provided by the system:

```
function EDGEPROG($vertexValue$,$edgeValue$,$vertexID$) :
    return $vertexID$
function VERTEXUPD($vertexValue1$,$vertexValue2$) :
    return $vertexValue1$
```

PageRank: PageRank is a good example of an algorithm with very dense active vertex sets. It also requires a finalize function for dampening, which is often defined as $x = 0.15/NumVertices + 0.85 * v$. In order to remove the performance effects of various algorithmic modifications to PageRank and do a fair comparison between systems, we only measured the performance of the very first iteration of PageRank, when all vertices are active.

PageRank can be expressed using the following program. *numNeighbors* is provided by the system:

```
function EDGEPROG($vertexValue$,$edgeValue$,$numNeighbor$) :
    return $vertexValue/numNeighbor$
function VERTEXUPD($vertexValue1$,$vertexValue2$) :
    return $vertexValue1+vertexValue2$
```

Betweenness-Centrality: BC is a good example of an algorithm that requires backtracing, which is an important tool for many applications including machine learning and bioinformatics. Each backtracking step incurs random updates to parent's vertex data, which is also handled with another execution of sort-reduce.

The edge and vertex programs for betweenness centrality is identical to BFS. After traversal is finished, each generated vertex list's vertex values are each vertex's parent vertex ID. Each list can be made ready for backtracing by taking the vertex values as keys and initializing vertex values to 1, and sort-reducing them. The backtrace sort-reduce algorithm can be expressed with the following algorithm:

```
function EDGEPROG($vertexValue$,$edgeValue$,$null$) :
    return $vertexValue$
function VERTEXPROG($vertexValue1$,$vertexValue2$) :
    return $vertexValue1+vertexValue2$
```

Once all lists have been reversed and reduced, the final result can be calculated by applying set union to all reversed vertex lists using a cascade of set union operations, with a custom function to add multiply values whenever there is a key match.

B. Graph Datasets

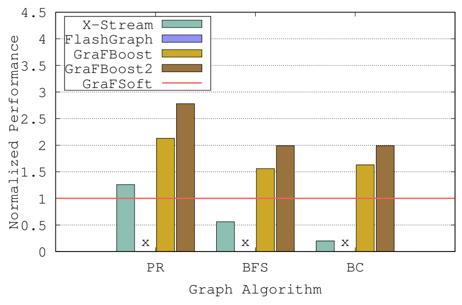
For each application, we used multiple different graph datasets with different sizes. One important graph dataset we analyzed is the Web Data Commons (WDC) web crawl graph [14] with over 3.5 billion vertices, adding up to over 2 TBs in text format. The WDC graph is one of the largest real-world graphs that is available. Others include kronecker graphs generated at various sizes according to Graph 500 configurations, and the popular twitter graph. Table I describes some of the graphs of interest. The size of the dataset is measured after column compressed binary encoding in GraFBoost's format.

C. Evaluation with Graph with Billions of Vertices

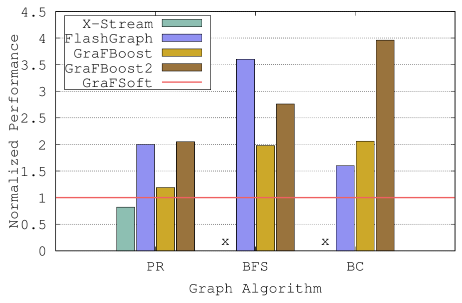
To show the effectiveness of the sort-reduce method in software (GraFSoft) and the hardware acceleration of that method (GraFBoost), we compared their performance

name	twitter	kron28	kron30	kron32	wdc
nodes	41M	268M	1B	4B	3B
edges	1.47B	4B	17B	32B	128B
edgefactor	36	16	16	8	43
size	6 GB	18 GB	72 GB	128 GB	502 GB
txtsize	25 GB	88 GB	351 GB	295 GB	2648 GB

Table I: Graph datasets that were examined



(a) Performance of the algorithms on the Kronecker 32 graph, on a machine with 128 GB of memory, normalized to the performance of software GraFBoost **(Higher is faster)**



(b) Performance of the algorithms on the WDC graph, on a machine with 128 GB of memory, normalized to the performance of software GraFBoost **(Higher is faster)**

Figure 12: Performance of graph algorithms on large graphs

against semi-external and external graph analytics frameworks for large graphs. To perform this evaluation, we are using a single server with 32 cores of Intel Xeon E5-2690 (2.90GHz) and 128 GB of DRAM. For graph storage, the server is equipped with five 512 GB PCIe SSDs with a total of 6 GB/s of sequential read bandwidth.

GraFBoost was implemented on the BlueDBM platform [72], where a hardware accelerated storage system was plugged into the PCIe slot of a server with 24-core Xeon X5670 and 48 GBs of memory. The storage device consists of a Xilinx VC707 FPGA development board equipped with 1 GB DDR3 SODIMM DRAM card, and augmented with two 512 GB NAND-flash expansion cards, adding up to 1 TB of capacity. Each flash expansion card is capable of delivering 1.2 GB/s read performance and 0.5 GB/s write performance, while the DRAM card can deliver 10 GB/s. We observed that the biggest performance bottleneck of GraFBoost is the single slow DIMM card, and also compare against the projected performance of GraFBoost2, equipped with DRAM capable of 20 GB/s.

To start the evaluation, we explore the performance of our three algorithms on a synthesized scale 32 kronecker graph, and the largest real-world graph we could find, the WDC web crawl graph.

1) Performance Overview: Figure 12a and Figure 12b shows the performance of various systems running graph algorithms on the Kron32 graph, and the WDC graph, respectively. All performance has been normalized to the performance of GraFSoft (red horizontal line).

For Kron32, 128 GB of memory was not enough for FlashGraph to fit all vertex data in memory, and did not finish for any algorithms. X-Stream was capable of running all algorithms to completion, but it performance was much slower than GraFBoost, and often even slower than GraFSoft. GraFBoost was able to deliver 57 MTEPS (Million Traversed Edges Per Second) on BFS.

The WDC graph has much fewer vertices than Kron32, and FlashGraph was often able deliver faster performance compared to GraFBoost, by accommodating all required data in memory. The relative performance of X-Stream for BFS and BC is too slow to be visible due to the limitations of the system in handling sparse active vertices. We observed each superstep taking about 500 seconds, which will result in the execution time exceeding two million seconds, or 23 days. On the other hand, BFS on kron32 required only 8 supersteps, resulting in much better performance on X-Strem (Figure 12a). GraFBoost delivered 75 MTEPS on BFS.

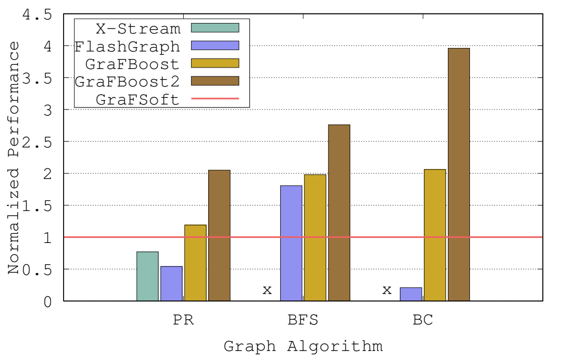
We were not able to measure the performance of GraphChi or GraphLab, due to low performance and insufficient memory, respectively.

It can be seen from the two large graphs that the GraFBoost systems are the only ones capable of delivering consistently high performance with terabyte-scale graphs on the given machine. This strength becomes even more pronounced when graphs are even larger relative to available memory size, which we will analyze in detail in the following section.

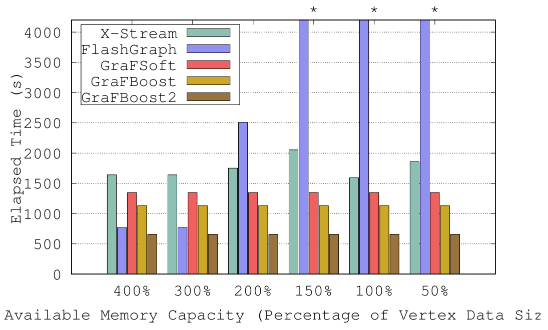
2) Performance with Larger Graphs Relative to Memory Capacity: We evaluated the performance impact of graph sizes relative to available memory, by processing large graphs on systems with variable memory capacities. As graph sizes relative to memory capacity became larger, the GraFBoost systems quickly became the only systems capable of maintaining high performance.

Figure 13a shows the performance of systems working on the WDC graph on a reasonably affordable machine with 32 Xeon cores and 64 GBs of memory (instead of 128 GB as before), as normalized to the performance of GraFSoft. We can see that the hardware accelerated GraFBoost implementations performs better than all systems compared, and even GraFSoft is faster than most systems.

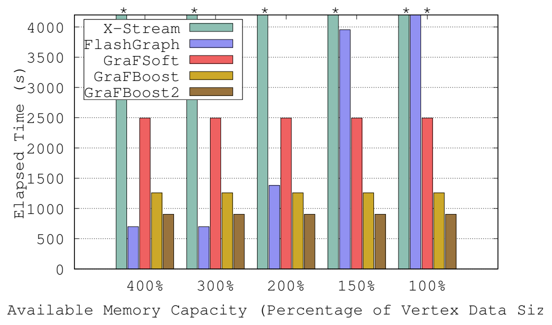
Below, we analyze the performance impact of memory



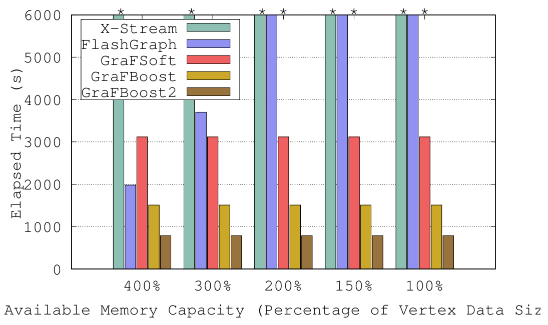
(a) Performance of the algorithms on a machine with 64 GB of memory, normalized to the performance of software GraFBoost **(Higher is faster)**



(b) PageRank iteration execution time on machines with various amounts of memory **(Lower is faster)**



(c) Breadth-First-Search execution time on machines with various amounts of memory **(Lower is faster)**



(d) Betweenness-Centrality execution time on machines with various amounts of memory **(Lower is faster)**

Figure 13: Performance of graph algorithms on the Web Data Commons graph, on systems with varying memory capacities

capacity for different graph algorithms. Memory capacity is denoted in percentage of vertex data size, where 100% is the space required to store 2^{32} 8-byte values, or 32 GB.

PageRank: Figure 13b shows the performance of each system running PageRank on the wdc graph on a system with various memory capacities. Performance of GraFBoost is consistently high regardless of memory capacity because its memory requirement is lower (16 GB or less for GraF-Soft, megabytes for GraFBoost).

FlashGraph showed fast performance with enough memory. But its performance degrades sharply and quickly becomes the slowest as the problem size becomes larger compared to memory capacity, causing swap thrashing. The execution instances marked with * were were stopped manually because it was taking too long.

X-Stream was the relatively slowest system with enough memory, but it was able to maintain performance with smaller memory by partitioning the graph into two and four partitions in 32 GB and 16 GB machines, respectively, in order to fit each partition in available memory. One caveat was that the vertex update logs between partitions became too large to fit in our flash array, and had to install more storage.

Breadth-First Search: Figure 13c shows the performance of breadth-first search on the wdc graph on various system configurations. Because the active vertex list during BFS execution is usually small relative to the graph size, the memory requirements of a semi-external system like FlashGraph is also relatively low. As a result, FlashGraph continues to demonstrate fast performance even on machines with relatively small memory capacity, before eventually starting thrashing.

We were unable to measure the performance of X-Stream on the wdc graph in a reasonable amount of time for any configuration. Execution of BFS on the WDC graph has a very long tail, where there were thousands of supersteps with only a handful of active vertices, which X-Stream is very bad for because it must process the whole graph for every iteration.

Betweenness-Centrality: Figure 13d shows the performance of betweenness-centrality on the wdc graph with FlashGraph and GraFBoost. The memory requirements of BC is higher than PageRank and BFS, resulting in faster performance degradation for FlashGraph. We were unable to measure the performance of X-Stream for BC on the wdc graph for the same reason with BFS, but it showed much better performance on the kron32 graph because of the low superstep count (Figure 12a).

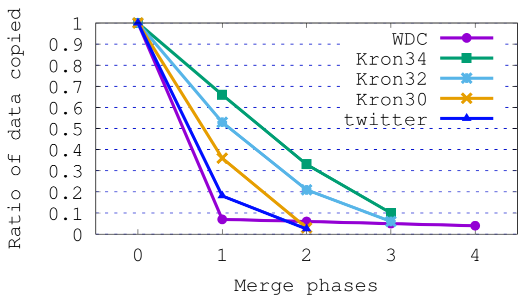


Figure 14: The fraction of data that is written to storage after each merge-reduce phase

3) Performance of Hardware Sort-Reduce: While the hardware sort-reduce accelerator provides superior performance compared to the software implementation, its most prominent performance bottleneck is the single instance of the in-memory sorter. By emitting 256 bit packed tuples every cycle, it can sustain 4 GB/s on our accelerator running at 125MHz, which almost saturates the bandwidth of our on-board DRAM. Our measurements showed sorting a single 512MB chunk took slightly over 0.5s. Furthermore, because the memory capacity is barely enough to facilitate 512MB sort, we cannot overlap graph access and in-memory sort. The only difference of the projected GraFBoost2 system compared to GraFBoost is double the DRAM bandwidth, achieving in-memory sort in a bit more than 0.25s.

Our software implementation of sort-reduce spawns up to four instances of the 16-to-1 merger, each emitting up to 800MB merged data per second. Spawning more software instances was not very effective because of computation resource limitations during in-memory sort, and flash bandwidth limitations during external sort. This performance difference can be seen in the PageRank example in Figure 13b, where graph adjacency list access is very regular and performance is determined by the performance of sort-reduce.

Performance difference becomes even more pronounced in the BFS and BC examples in Figure 13c and Figure 13d. This is due to the latency difference of off-the-shelf SSDs and GraFBoost hardware with a lower-overhead flash management layer. Thanks to the low access latency, hardware GraFBoost is able to maintain performance even with small lookahead buffers, which almost removes unused flash reads.

4) System Resource Utilization: Table II shows the typical system resource utilizations of the systems compared, while running at full capacity PageRank on the WDC graph. Both FlashGraph and X-Stream attempted to use all of the available 32 cores' CPU resources. It is most likely because these systems spawn a lot more threads than GraFBoost. As a result, both systems record 3200% CPU usage while running. The software GraFBoost tries to use all of the available processing resources, but does not exceed 1800% because performance is usually bound by storage performance. The hardware accelerated GraFBoost does not require much processor resources because the resource-intensive sort-reduce has been offloaded to the hardware accelerator.

name	Memory	Flash Bandwidth	CPU
GraFBoost	2 GB	1.2 GB/s	200%
GraFSoft	8 GB	500 MB/s*, 4 GB/s**	1800%
FlashGraph	60 GB	1.5 GB/s	3200%
X-Stream	80 GB	2 GB/s	3200%

*During intermediate list generation
**During external merge-reduce

Table II: Typical system resource utilization during PageRank on WDC

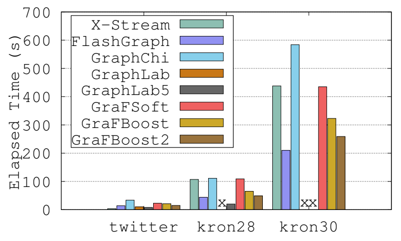
5) Benefits of Interleaving Reduction with Sorting: One of the big factors of GraFBoost's high performance is the effectiveness of interleaving reduction operations with sorting. Figure 14 shows the ratio of data that was merge-reduced at every phase compared to if reduction was not applied, starting from the intermediate list generated when all nodes are active. The number of merge-reduce phases required until the intermediate data is completely merged depends on the size of the graph.

The reduction of data movement is significant, especially in the case of the two real world graphs, the twitter graph and WDC graph. In these graphs, the size of the intermediate list has already been reduced by over 80% and 90% even before the first write to flash. This reduces the amount of total writes to flash by over 90%, minimizing the impact of sorting and improving flash lifetime.

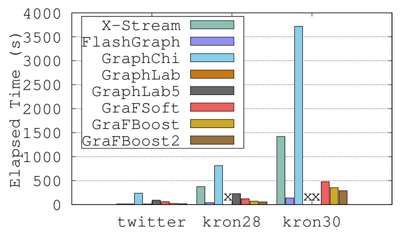
6) Power Consumption: One major advantage of an accelerated storage device is the lowered power consumption due to offloading computation to the accelerator. Our GraFBoost prototype consumes about 160W of power, of which 110W is consumed by the host Xeon server which is under a very low load. The host server could conceivably be replaced with a lower-power embedded server, which will greatly reduce the power consumption without sacrificing performance. For example, with a wimpy server with a 30W power budget will bring down its power consumption to half, or 80W. This is in stark contrast, for example, to our setup running FlashGraph, which was consuming over 410W during operation. The power consumption of flash storage is relatively low, as each SSD typically consumes less than 6W. Because FlashGraph is also computation intensive, moving it to weaker servers would directly result in performance loss.

D. Evaluation with Small Graphs (Less Than 1 Billion Vertices)

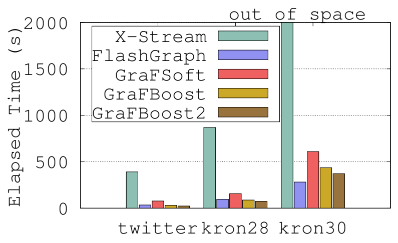
The three figures in Figure 15 shows the processing time for PageRank, BFS and BC on smaller graphs with 1 billion or less vertices. Such small graphs are not the target application of GraFBoost, but are provided for insight. Software evaluations were done using the same server with



(a) Execution time of PageRank on small graphs **(Lower is faster)**



(b) Execution time of breadth first search on small graphs **(Lower is faster)**



(c) Execution time of betweenness centrality on small graphs **(Lower is faster)**

Figure 15: Execution time of graph algorithms on small graphs (On a single server with one SSD)

32 cores and 128 GB of memory, but were only provided with one SSD instead of five. GraFBoost also used only one flash card instead of one, matching 512 GB capacity and 1.2GB/s bandwidth. At the risk of doing an unfair comparison, we also evaluated a 5-node cluster of distributed GraphLab (GraphLab5) using systems with 24-core Xeon X5670 and 48 GB of memory networked over 1G Ethernet.

GraphLab cannot handle graphs larger than the twitter graph, and GraphLab5 cannot handle graphs larger than Kron28. While GraphLab5 demonstrates the best performance for PageRank on Kron28, it is relatively slow for BFS, even against single-node GraphLab for the twitter data. This is most likely due to the network becoming the bottleneck with irregular data transfer patterns.

For small graphs, the relative performance of GraphBoost systems are not as good as with bigger graphs, but demonstrates comparable performance to the fastest systems. When datasets are small, the effectiveness of cacheing graph edge data in semi-external systems increases, and sort-reduce becomes an unnecessary overhead.

VI. Conclusion

In this paper, we have presented GraFBoost, a external graph analytics system with hardware-accelerated flash storage that uses a novel sort-reduce accelerator to perform high-speed analytics on graphs with billions of vertices, on an affordable machine with very small memory. A wimpy PC-class machine coupled with a GraFBoost accelerated storage could achieve server-class performance with a much lower power footprint. A GraFBoost accelerated storage device could be packaged into a programmable SSD device that can be simply plugged into a machine to achieve high performance graph analytics.

We are also actively working on scalable GraFBoost. For much larger graphs, GraFBoost can easily be scaled horizontally simply by plugging in more accelerated storage devices into the host server. The intermediate update list can be transparently partitioned across devices using BlueDBM's inter-controller network. For even larger graphs, GraFBoost can easily be scaled to run on a cluster with the same method.

We have implemented GraFBoost to use a sparse adjacency representation of graphs, which is not well suited for dynamically changing graphs. However, any graph structure representation can be used with the GraFBoost accelerated storage to remove random access.

Furthermore, the sort-reduce accelerator is generic enough to be useful beyond graph analytics. We plan to explore various applications to develop a more general environment for external computation.

Acknowledgements. This work was partially funded by NSF(CCF-1725303) and Samsung(GRO grants 2015 and 2017).

References

[1] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *KDD*. New York, NY, USA: ACM, 2003.
[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *KDD*. ACM, 2006.
[3] J. W. Lichtman, H. Pfister, and N. Shavit, "The big data challenges of connectomics," *Nature neuroscience*, vol. 17, no. 11, pp. 1448–1454, 2014.
[4] X. Fang, S. Misra, G. Xue, and D. Yang, "Smart grid - the new and improved power grid: A survey," *IEEE Communications Surveys Tutorials*, vol. 14, no. 4, April 2012.
[5] J. Xu and H. Chen, "Criminal network analysis and visualization," *Communications of the ACM*, vol. 48, no. 6, pp. 100–107, 2005.
[6] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar *et al.*, "The case for ramcloud," *Communications of the ACM*, vol. 54, no. 7, pp. 121–130, 2011.
[7] 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*, 2015.
[8] 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*. ACM, 2013.
[9] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *SIGMOD*. New York, NY, USA: ACM, 2013.
[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, Apr. 2012.
[11] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A new framework for parallel machine learning," in *UAI*, 2010.
[12] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *SOSP*. ACM, 2013.
[13] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a pc," in *OSDI*. Hollywood, CA: USENIX, 2012.
[14] "Web data commons - hyperlink graphs," http://webdatacommons.org/hyperlinkgraph/, accessed: 2017-10-01.
[15] "Brief introduction — graph 500," http://www.graph500.org/, accessed: 2017-11-17.
[16] "NVMe SSD 960 PRO EVO brochure," http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/NVMe_SSD_960_PRO_EVO_Brochure.pdf, accessed: 2017-11-21.
[17] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[18] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *EuroSys*. ACM, 2013.
[19] Y. Zhao, K. Yoshigoe, M. Xie, S. Zhou, R. Seker, and J. Bian, "Lightgraph: Lighten communication in distributed graph-parallel processing," in *Big Data (BigData Congress)*. IEEE, 2014.
[20] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan, "Replication-based fault-tolerance for large-scale graph processing," in *DSN*. IEEE, 2014.
[21] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013.
[22] 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*. ACM, 2010.
[23] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Hadoop Summit*, vol. 11, no. 3, 2011.
[24] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, Oct. 2015.
[25] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *SOSP*. New York, NY, USA: ACM, 2013.
[26] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, 2013.
[27] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.
[28] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *USENIX ATC*. Santa Clara, CA: USENIX Association, 2015.
[29] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, 2011.
[30] J. Kepner, D. Bader, A. Buluç, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, matrices, and the graphblas: Seven good reasons," *Procedia Computer Science*, vol. 51, pp. 2453–2462, 2015.
[31] D. Hutchison, J. Kepner, V. Gadepally, and A. Fuchs, "Graphulo implementation of server-side sparse matrix multiply in the Accumulo database," in *HPEC*. IEEE, 2015.
[32] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011, vol. 22.
[33] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 22.
[34] S. Zhou *et al.*, "Gemini: Graph estimation with matrix variate normal instances," *The Annals of Statistics*, vol. 42, no. 2, pp. 532–562, 2014.
[35] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012.
[36] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic i/o optimization for disk-based graph processing," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 507–522. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/vora
[37] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 363–374.
[38] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang, "Mmap: Fast billion-scale graph computation on a pc via memory mapping," in *Big Data*, Oct 2014.
[39] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *SIGPLAN Not.*, vol. 48, no. 8, Feb. 2013.
[40] "Neo4j: The world's leading graph databse," https://neo4j.com/, accessed: 2017-11-17.
[41] J. Fan, A. G. S. Raj, and J. M. Patel, "The case against specialized graph analytics engines." in *CIDR*, 2015.
[42] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *SIGMETRICS*. New York, NY, USA: ACM, 2009.
[43] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *Journal of Systems Architecture*, vol. 55, no. 5, pp. 332–343, 2009.
[44] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.
[45] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *FAST*. USENIX Association, 2015.
[46] A. One, "Yaffs: Yet another flash file system," 2002.
[47] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann, "Noftl: Database systems on ftl-less flash storage," *Proceedings of the VLDB Endowment*, vol. 6, no. 12, pp. 1278–1281, 2013.
[48] S. Lee, M. Liu, S. W. Jun, S. Xu, J. Kim, and Arvind, "Application-managed flash," in *FAST*. USENIX Association, 2016, pp. 339–353.
[49] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *MICRO*. IEEE Computer Society, 2010.
[50] A. M. Caulfield and S. Swanson, "Quicksan: a storage area network for fast, distributed, solid state disks," in *ISCA*. ACM, 2013, pp. 464–474.
[51] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 365–376.
[52] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *ASPLOS*. New York, NY, USA: ACM, 2014.
[53] E. S. Chung, J. D. Davis, and J. Lee, "Linqits: Big data on little clients," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 261–272.
[54] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, "Database analytics acceleration using fpgas," in *PACT*. ACM, 2012.
[55] L. Woods, Z. Istvan, and G. Alonso, "Hybrid fpga-accelerated sql query processing," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–1.
[56] "Graph CREST," http://opt.imi.kyushu-u.ac.jp/graphcrest/eng/, accessed: 2017-11-21.
[57] 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*. ACM, 2016.
[58] 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*. IEEE, 2016.
[59] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*. IEEE, 2015.
[60] X. Ma, D. Zhang, and D. Chiou, "Fpga-accelerated transactional execution of graph workloads," in *FPGA*. New York, NY, USA: ACM, 2017.
[61] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "Unlocking ordered parallelism with the swarm architecture," *IEEE Micro*, vol. 36, no. 3, pp. 105–117, 2016.
[62] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "GraphGen: An FPGA framework for vertex-centric graph computation," in *FCCM*. IEEE, 2014.
[63] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," in *FPGA*. New York, NY, USA: ACM, 2017.
[64] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *PPoPP*. New York, NY, USA: ACM, 2016.
[65] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent distributed storage," *Proc. VLDB Endow.*, vol. 10, no. 11, Aug. 2017.
[66] 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*. Piscataway, NJ, USA: IEEE Press, 2016.
[67] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: A case for intelligent ssds," in *ICS*. ACM, 2013.
[68] G. Koo, K. K. Matam, T. I, H. V. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: Trading communication with computing near storage," in *MICRO*. New York, NY, USA: ACM, 2017.
[69] 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*. ACM, 2013.
[70] S. Seshadri, M. Gahagan, M. S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmable ssd." in *OSDI*, 2014, pp. 67–80.
[71] "Project catapult - microsoft research," https://www.microsoft.com/en-us/research/project/project-catapult/, accessed: 2017-11-17.
[72] 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*. ACM, 2015.
[73] B. Y. Cho, W. S. Jeong, D. Oh, and W. W. Ro, "Xsd: Accelerating mapreduce by harnessing the gpu inside an ssd," in *Proceedings of the 1st Workshop on Near-Data Processing*, 2013.
[74] M. Oskin, F. T. Chong, and T. Sherwood, *Active pages: A computation model for intelligent memory*. IEEE Computer Society, 1998, vol. 26, no. 3.
[75] E. S. Chung, J. C. Hoe, and K. Mai, "Coram: An in-fabric memory architecture for fpga-based computing," in *FPGA*, 2011.
[76] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *EuroSys*. ACM, 2017.
[77] M. Liu, S.-W. Jun, S. Lee, J. Hicks *et al.*, "minflash: A minimalistic clustered flash array," in *DATE*. IEEE, 2016, pp. 1255–1260.
[78] S. W. Jun, S. Xu, and Arvind, "Terabyte sort on fpga-accelerated flash storage," in *FCCM*, April 2017.