GraphSSD: Graph Semantics Aware SSD

Kiran Kumar Matam
kmatam@usc.edu
University of Southern California
Los Angeles, California

Gunjae Koo
gunjae.koo@hongik.ac.kr
Hongik University
Seoul, South Korea

Haipeng Zha
hzha@usc.edu
University of Southern California
Los Angeles, California

Hung-Wei Tseng
htseng3@ncsu.edu
North Carolina State University
Raleigh, North Carolina

Murali Annavaram
annavara@usc.edu
University of Southern California
Los Angeles, California

ABSTRACT

Graph analytics play a key role in a number of applications such as social networks, drug discovery, and recommendation systems. Given the large size of graphs that may exceed the capacity of the main memory, application performance is bounded by storage access time. Out-of-core graph processing frameworks try to tackle this storage access bottleneck through techniques such as graph sharding, and sub-graph partitioning. Even with these techniques, the need to access data across different graph shards or sub-graphs causes storage systems to become a significant performance hurdle. In this paper, we propose a graph semantic aware solid state drive (SSD) framework, called GraphSSD, which is a full system solution for storing, accessing, and performing graph analytics on SSDs. Rather than treating storage as a collection of blocks, GraphSSD considers graph structure while deciding on graph layout, access, and update mechanisms. GraphSSD replaces the conventional logical to physical page mapping mechanism in an SSD with a novel vertex-to-page mapping scheme and exploits the detailed knowledge of the flash properties to minimize page accesses. GraphSSD also supports efficient graph updates (vertex and edge modifications) by minimizing unnecessary page movement overheads. GraphSSD provides a simple programming interface that enables application developers to access graphs as native data in their applications, thereby simplifying the code development. It also augments the NVMe (non-volatile memory express) interface with a minimal set of changes to map the graph access APIs to appropriate storage access mechanisms.

Our evaluation results show that the GraphSSD framework improves the performance by up to 1.85× for the basic graph data fetch functions and on average 1.40×, 1.42×, 1.60×, 1.56×, and 1.29× for the widely used breadth-first search, connected components, random-walk, maximal independent set, and page rank applications, respectively.

CCS CONCEPTS

• **Hardware** → **External storage**.

KEYWORDS

SSD, Graphs, Flash storage

ACM Reference Format:
Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: Graph Semantics Aware SSD. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307650.3322275

1 INTRODUCTION

Graphs analytics are at the heart of a broad range of applications such as social network analysis, drug discovery, page ranking, transportation systems, and recommendation systems. The size of the graphs in many of these domains exceeds the size of main memory seen in commodity computing systems. There is a need to consider efficient storage-centric graph processing, at least in a subset of application scenarios where the size of the graph far exceeds the size of the main memory. It is well known that data input/output (I/O) time to access large graphs consumes a significant fraction of the total execution time compared to the CPU and memory access time [2, 10, 32].

On the storage front, the cost of solid-state drives (SSDs) has fallen dramatically. NAND Flash SSDs cost about $100 per 1TB as of early 2019, and the price is expected to reduce further. With the advent of non-volatile memory express (NVMe) [13] interface, SSDs can offer significant improvements in bandwidth and enable tighter integration of computing with storage. Furthermore, SSDs are equipped with reasonably capable compute fabric to handle flash management tasks. The advent of such affordable SSDs provides new opportunities to improve the performance of graph analytics by making storage systems semantically aware of the graph data being stored. In particular, we make a case for treating graphs as a native format supported on storage, rather than treating graphs as a collection of pages that are accessed using standard block I/O interface.

This work presents the design and implementation of a graph semantic aware SSD (GraphSSD) to manage graphs on an SSD platform. GraphSSD supports the compressed sparse row (CSR) format for graph layout and further customizes this format to enable fast mapping of vertex id to the physical page location that contains the adjacency information of that vertex. GraphSSD provides a set of programming APIs to application developers to access graph vertex and edge information, similar to recent graph frameworks such as GraphCHI [21] and Pregel [27]. But the novelty of this work is that the SSD controller is made aware of the graph data structures stored on the SSD. Thus the controller can automatically translate the graph access APIs into a set of low level physical page accesses to fetch the requested data. These APIs accept basic graph related queries such as fetching adjacent vertices for a target vertex, fetching edge weights of connected edges. GraphSSD APIs are generic enough to enable developers to write complex graph analytics on top of GraphSSD. GraphSSD provides solutions to the following challenges:

(1) Graph as native objects: GraphSSD allows the embedded controller in SSDs to treat graphs as native storage objects, and provides a set of APIs that can be used to access the graph objects on storage.
(2) NAND flash aware graph layout: NAND flash memory can only be accessed in fixed-size pages, and pages can be accessed concurrently only across the parallel units. The widely varying sizes of the adjacent neighbors across different vertices require a graph storage mechanism that accommodates this diversity within the constraints of the flash memories. GraphSSD tackles this challenge by relying on a compressed sparse row (CSR) representation for a graph, and embedding metadata in NAND pages to store edges from one or more vertices.
(3) Efficient indexing mechanism: GraphSSD presents an innovative graph translation layer (GTL), which translates a vertex id to a physical page address on the flash memory media directly, thereby reducing unnecessary indirect page accesses to reach a given vertex.
(4) Indexing compaction: GraphSSD reduces the GTL mapping overhead by co-locating multiple vertices with few edges in the same physical page.
(5) Support for graph updates: GraphSSD relies on Delta graphs and Delta merging mechanisms that allow GraphSSD to modify only a small subset of pages containing the updated subgraph instead of re-shuffling the entire graph.
(6) We implement GraphSSD framework on an industrial strength SSD development platform to show the performance improvement of GraphSSD over a conventional graph storage architecture. Our evaluation results show that the GraphSSD framework improves the performance by up to 1.85× for the basic graph data fetch functions and on average 1.40×, 1.42×, 1.60×, 1.56×, and 1.29× for the widely used breadth-first search, connected components, random-walk, maximal independent set, and page rank applications, respectively.

The remainder of this paper is organized as follows: Section 2 introduces the operations of an SSD platform, the graph storage format as a background and motivates the need for a graph semantic aware storage device. The detailed architecture and functions of GraphSSD are described in Section 3. The implementation methodology, evaluation platform and the experimental results are presented in Sections 4, 5, and 6 respectively. Related work is provided in Section 7, and we conclude in Section 8.

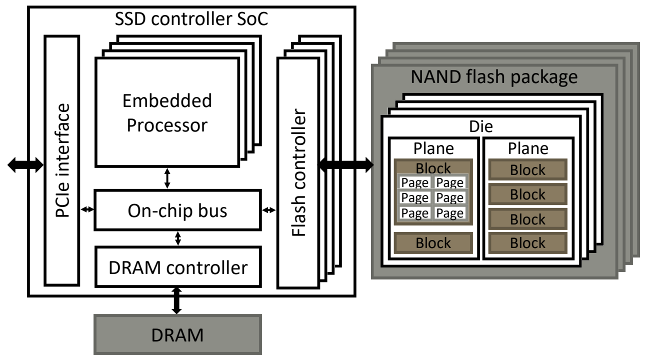


Figure 1: Modern SSD platform architecture

2 BACKGROUND AND CHALLENGES

2.1 Modern SSD platforms

Figure 1 illustrates the architecture of a modern SSD platform. An SSD equips multiple flash memory channels to support high data bandwidth. Multiple dies are integrated into a single NAND flash package by employing die-stacking structure to integrate more storage space on the limited platform board. Data parallelism can be achieved per die with multi-plane or multi-way composition. Each plane or way is divided into multiple blocks which have dozens of physical pages.

A page is a basic physical storage unit that can be read or written by one flash command. The page size has steadily increased as more flash memory cells can be integrated within the same area by using multi-bit or vertical cell technology [18]. Unlike the magnetic storage devices, flash memory cells need to be initialized before a write operation. This erasure process can be performed only at a block granularity since erasing flash memory cells requires higher electric energy, which can pollute neighboring page cells. In addition, the erasure process is significantly slower than reading or writing. Thus SSDs write the updated page contents to a new *empty* physical page rather than *erase-and-write* an entire block. Consequently, the logical block address (LBA) of a flash page is mapped to new physical page address (PPA) in the flash memory space whenever the page data is updated (and the old page is invalidated). The SSD controller firmware manages this mapping information in the flash translation layer (FTL) mapping table. The SSD controller does garbage collection of invalid pages to create empty physical pages. It erases an entire block, writing any valid physical pages in that block to another empty physical page in a different block and updating the FTL.

2.2 Out-of-core graph processing

In many domains, graphs are large requiring out-of-core graph processing. Namely, graphs are processed in smaller chunks where each chunk is read from the storage into the DRAM. Large graphs also tend to be sparsely connected and hence to reduce the I/O bottleneck large graphs are stored in a compressed format, such as the compressed sparse row (CSR) format. Alternate approaches have also been proposed to access graphs in smaller chunks such as shards [21]. Irrespective of the choice of the graph storage format, all prior techniques require the storage to be treated as a block device. In this paper, the storage controller understands the semantics of graphs

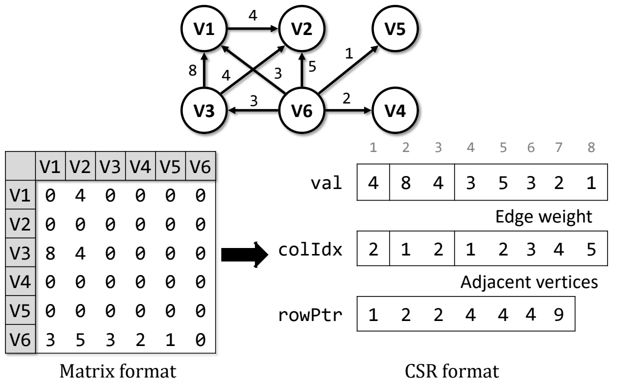


Figure 2: CSR format representation for an example graph

and the system provides a set of APIs to program the storage controller to access graph data. In this context we use CSR format in our implementation. CSR format is widely used for general purpose large-scale graph processing [30, 41] and is shown to be efficient for out-of-core graph processing systems [16, 22, 26, 30].

CSR format takes the adjacency matrix representation of a graph and compresses it using three vectors. The value vector, *val*, stores all the non-zero values from each column sequentially. The column index vector, *colIdx*, stores the column index of each element in the *val* vector. The row pointer vector, *rowPtr*, stores the starting index of each row in the *val* vector. CSR format representation for an example graph is shown in Figure 2.

To access adjacent vertices associated with a vertex in the CSR graph storage format, we first need to access the *rowPtr* vector to get the starting index in the *colIdx* vector. The *colIdx* vector stores the adjacent vertices associated with the vertex in a contiguous fashion. To get an edge weight, we need to get the adjacent vertices for the source vertex and find the index of the destination vertex in the *colIdx* vector; with that index we can access the corresponding location in the *val* vector, where edge weights are stored.

2.3 Graph updates

Many of the existing graph frameworks assume static graphs but there is a need to support dynamic graphs whose edge and vertex related information may be updated [30]. GraphSSD supports efficient graph updates by building on Delta graphs introduced in LLAMA [26]. GraphSSD maintains graph updates as a series of snapshots. Initially, all the vertices store their adjacency list as one contiguous vector. When graph updates are performed, multiple updates are grouped together into a single snapshot. At regular intervals a snapshot is written back to storage and a new snapshot is created for the incoming updates. Multiple snapshots are chained together alongside the initial adjacency list as a linked list of pointers. To retrieve the entire adjacency list for a vertex, one has to traverse these chains of pointers starting from newer snapshots to older snapshots. This pointer chasing may increase the latency for accessing adjacent vertices. To reduce this inefficiency GraphSSD also merges the snapshots at regular intervals to create a single contiguous adjacency list for each vertex.

When using a CSR based graph representation to merge updates one has to read the entire graph and merge the updates and write to a new location [26, 30]. To have a consistent view one can read a vertex's adjacency list from the new location only after the entire merge operation has been performed. During the long merging process, there is a significant performance penalty to access the adjacency list for a vertex.

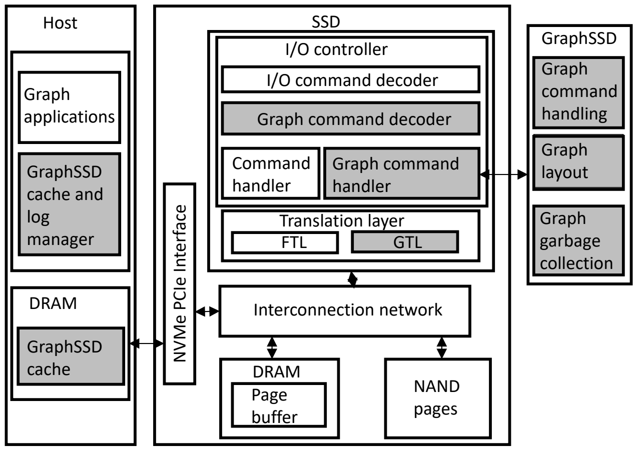


Figure 3: GraphSSD architecture overview

3 ARCHITECTURE

Figure 3 provides an overview of the GraphSSD architecture. GraphSSD has the following components: graph translation layer (GTL) and graph command decoder on the SSD; and a host side graph caching layer and a graph update logger.

3.1 Graph command decoder:

The capabilities of GraphSSD are exposed to the application programmer through a set of graph access APIs. These APIs are implemented within the GraphSSD library that can be linked into any graph application. Each of these APIs in turn activate the SSD microcontroller to perform some of the storage access tasks. These activation commands are transferred as NVMe commands issued from the host to the SSD. The current NVMe protocol has several unused bytes in the read/write command encoding which can be easily adapted to implement the GraphSSD APIs, if deemed necessary to minimize any protocol changes. All NVMe commands from the host are first intercepted by the GraphSSD command decoder. The command decoder routes all GraphSSD APIs to the graph command processing, while regular NVMe commands (page read and write) are sent to the default SSD processing sequence. Using this approach GraphSSD can co-exist with traditional block based storage interface within a single SSD.

Category	Commands
Graph read commands	GetAdjacencyList, GetEdgeWeight
Graph update commands	AddEdge, AddVertex, DeleteEdge, DeleteVertex, UpdateEdge, UpdateVertex
Graph initialization	GraphInitialize

Table 1: GraphSSD APIs

The list of APIs supported by GraphSSD currently is shown in Table 1. All GraphSSD APIs provide a vertex id to start any type of graph access. As such the first step in GraphSSD processing is to access the vertex id and its associated edges in the graph from the flash storage. To enable fast access to the physical pages consisting of the vertex related data we propose a novel graph translation layer as a substitute for traditional FTL used in SSDs.

3.2 Graph translation layer

In order to understand the workflow of GraphSSD it is important to understand how graphs are laid out in storage by our design. As described earlier, we assume that graphs are represented in CSR format using three vectors, named *rowPtr*, *colIdx*, and *val*. Each entry in the *rowPtr* vector is essentially the starting index into *colIdx* (and *val*) vectors where the neighbors of that vertex are located. GraphSSD essentially preserves this indexing mechanism while laying out these vectors in the flash pages. The *colIdx* and *val* vectors are proportional to the number of edges in the graph and hence they are significantly larger than the *rowPtr* vector. GraphSSD stores only the *colIdx* and *val* vectors in the NAND flash pages, and uses the *rowPtr* vector as an indexing table. GraphSSD provides a translation layer for this indexing purpose, called the Graph Translation Layer (GTL). GTL replaces the more traditional LBA-to-PPN (logical block address to physical page number) page mapping used in commodity SSDs. GraphSSD maps a given vertex id (V_i) to the physical NAND page number where the neighbor vertices (*colIdx* values) are stored.

GTL architecture: Figure 5 shows the structure of graph translation table (GTT) and Figure 4 shows the page layout for the graph. Each entry in GTT includes the mapping from a vertex id to the physical page number (PPN), and a tuple of status flags (dirty, extension, and valid). We will later describe how the GTT status flags are utilized. While conceptually each vertex maps to the physical pages storing all its neighbors through GTT, most of the real-world graphs have sparse connectivity. Hence, lots of vertices have only a few edges. As such, it is possible to co-locate the neighbors of multiple vertices in a single physical page. In this scenario, it is wasteful to allocate one GTT entry per vertex. To reduce this wastage GTT stores only one vertex id per physical page. GTT stores just the smallest vertex id from all the vertices whose neighbors are stored in a given physical page. Namely for each vertex V_i, there is a GTT entry indexed with vertex id V_j that is smaller than or equal to V_i. The next entry in GTT has a vertex id V_k which is greater than V_i. To make this search process efficient, GTL stores all the vertex ids in sorted order in GTT. If the graphs are directional, GraphSSD stores the incoming and outgoing edge information in separate pages, and keeps separate GTT for each of incoming and outgoing edge information.

Since each physical page may store neighbors of multiple vertices we need to identify the offset of the neighbor list for each vertex id. For this purpose, each physical page includes additional fields to store layout information as shown in Figure 4. We will describe the fields in the page starting from the last field and moving to the front. The last field in the physical page stores the number of vertices whose neighbor lists are stored in that page. Preceding this count there are N+1 <vertex, offset> tuples, corresponding to the N vertices stored in that page. Each tuple stores the vertex id and the starting byte offset within the page where the neighbor list for that vertex is stored in that page. Since the last adjacency list stored in page may fill a page partially a special tuple is used to indicate the ending offset for the last neighbor list. This ending offset of the last neighbor list is necessary to mark where the valid data in a page ends. The offset information and metadata described above is stored from the end of the page, and the actual vertex neighbor lists are stored from the starting of the page. Storing location pointers along with the adjacent vertices in a page helps us 1) in reducing the size of the GTT which will enable keeping a large chunk of GTT in DRAM, and 2) making no extra NAND page accesses to reach the adjacent vertices associated with a vertex id.

We now discuss different graph layout scenarios in GTT.

1. When neighbor vertices of a vertex V_i are stored entirely in a page: In this case, all neighbors are stored contiguously in the page and the starting offset of the neighbor list is stored in the location tuple associated with V_i.

2. When neighbor vertices of a vertex V_i are stored across multiple pages: V_i's neighbors span multiple pages for two reasons. First, V_i has many neighbors which will not fit in a single page. In this case, at least one page stores only the neighbors of V_i. That page will store just a single location pointer tuple and the last field on the page indicates that only a single vertex's neighbors are located on that page. After filling multiple full pages for a long neighbor list, there may be at most one partial page to store the last remaining neighbors. That page may also store the neighbors of other vertices. In this case, the location pointer tuples of all vertices including V_i are stored just as the first case above. GTL handles these dense vertices by storing the V_i to physical page mapping in GTT for each page that stores the neighbors of V_i. Thus V_i may have more than one GTT entry.

3. There is a third case where the number of neighbors of V_i may not fit in the existing free space in a page, and hence may span two different pages, even though the total number of vertices do not exceed a single page. We explored different options for packing the page but in the end, for simplicity of design, we decided to avoid spanning neighbors across two pages. Hence, if the neighbors do not fit in the leftover space in a page we simply allocate the neighbors to a new page.

An example of page layout: Figure 6 shows an example of GTT and the corresponding physical page layout. In this example, the GTL entry corresponding to $V1$ stores $P1$, indicating that the neighbors of $V1$ are stored in physical page $P1$. Since the next entry of GTL corresponds to vertex $V3$ it implies that the previous GTL entry also stores neighbors of $V2$. Similarly, second GTL entry shows that the neighbors of $V3$, $V4$ and $V5$ are stored in the physical page $P2$. Finally the neighbors of $V6$ span two physical pages $P3$ and $P4$.

The physical page organization is shown on the right half of Figure 6. For instance, the physical page $P2$ stores neighbors of three vertices and hence the last field (labeled $No.V$) shows the count to be 3. To the right of this field are the tuples that shows each vertex and its starting byte offset in the page. The tuple $(V3, 1)$ in physical page $P2$ shows that vertex $V3$'s neighbors are located starting at byte offset 1 within the page. A custom tuple $(End, 3)$ shows that the last valid byte on this page is byte 2. Hence GraphSSD can extract the neighbors of each vertex by decoding the graph page layout as described. The GTL also shows that V6 has two page entries since it is a dense vertex with many neighbors that span more than one page.

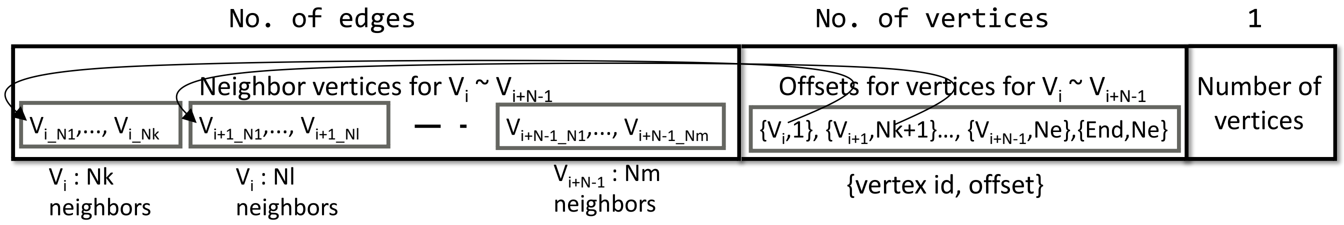

No. of edges
No. of vertices
1
Neighbor vertices for V_i ~ V_{i+N-1}
V_{i_N1},…, V_{i_Nk}
V_{i+1_N1},…, V_{i+1_Nl}
— -
V_{i+N-1_N1},…, V_{i+N-1_Nm}
Offsets for vertices for V_i ~ V_{i+N-1}
{V_i,1}, {V_{i+1},Nk+1}…, {V_{i+N-1},Ne},{End,Ne}
Number of vertices
V_i : Nk neighbors
V_i : Nl neighbors
V_{i+N-1} : Nm neighbors
{vertex id, offset}

Figure 4: Page layout

Vertex id	PPN	Dirty	Extension	Valid
V_i	PPN_i	0	0	1
V_{i+N}	PPN_j	0	0	1

Figure 5: Graph translation table

Vid	PPN	D	E	V
V1	P1	0	0	1
V3	P2	0	0	1
V6	P3	0	0	1
V6	P4	0	0	1

Graph translation table

PPN	1	2	3	4	5	6	No.V
P1	V2			{End,2}	{V2,2}	{V1,1}	2
P2	V1	V2	{End,3}	{V5,3}	{V4,3}	{V3,1}	3
P3	V1	V2	V3	V4	{End,5}	{V6,1}	1
P4	V5				{End,2}	{V6,1}	1

Page layout

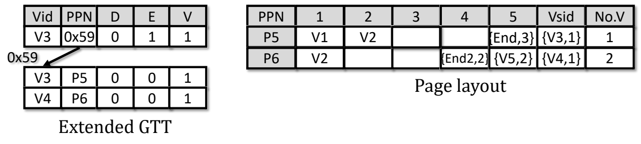
Figure 6: An example of GTT and page layout

Translation using GTT: Now we will discuss how GTL accesses the neighbors of a vertex id V_i using the GTT. For the sake of simplicity, assume that the graph is laid out initially on the NAND page as described in the example above and no updates have been made. GTL does a binary search on V_i column in GTT; recall the GTT entries are sorted and hence binary search is efficient. For a vertex V_i, GTL identifies indices j and k such that $j <= i <= k$. After that GTL fetches the page pointed by the V_j, say PPN_j. It then does a binary search on the location tuples in the page to match V_i in a tuple. If a match is found then the offset associated with V_i is then used to access the neighbor list. If V_i is a dense vertex that spans multiple pages then there will be multiple GTT entries for V_i. Hence GTL will access each of these pages to construct the neighbor list.

An example translation using GTT: Here we will discuss an example translation considering the example graph shown in Figure 6. We consider process of locating $V3$'s neighbors. First, to identify the pages in which $V3$'s adjacent vertices are stored, GTL does a binary search on vertex id's in GTT. As $V3 <= V3 < V6$, $V3$'s adjacent vertices are stored in the page pointed by the GTT entry with vertex id $V3$, i.e. $P2$. Then $P2$ page is fetched. The last field in $P2$ indicates that there are 3 vertices whose neighbors are stored in this page. Then GTL searches the neighbor tuples to identify where $V3$'s offset is, which is ($V3$,1). This location pointer indicates that adjacent vertices for $V3$ can be found in $P2$ page at offset starting from 1. Based on next tuple's ($V4$,3) offset, GTL identifies the size of $V3$'s neighbors list as 2.

3.3 Supporting graph updates

In this section we briefly discuss GTT support for operations that modify the graph: namely *AddEdge* - add an edge between two vertices and also edge weight for it, *AddVertex* - which adds a new vertex and a list of its adjacent vertices edge weights. Here we will describe how each of these operations updates the graph data on the NAND pages and the GTT.



Vid	PPN	D	E	V
V3	0x59	0	1	1

0x59

V3	P5	0	0	1
V4	P6	0	0	1

Extended GTT

PPN	1	2	3	4	5	Vsid	No.V
P5	V1	V2			{End,3}	{V3,1}	1
P6	V2			{End2,2}	{V5,2}	{V4,1}	2

Page layout

Figure 7: An example for updating GTT with extended bit

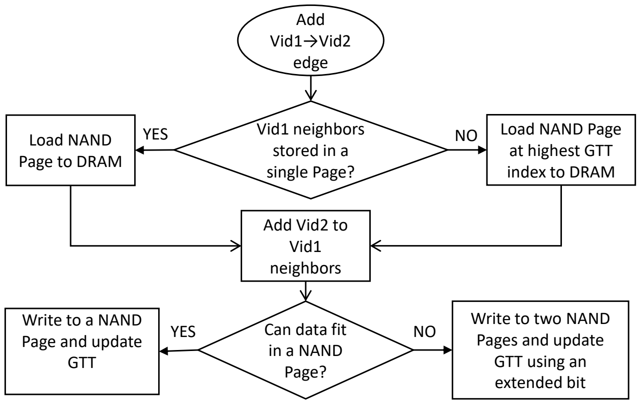

Add Vid1→Vid2 edge
Vid1 neighbors stored in a single Page?
YES
Load NAND Page to DRAM
NO
Load NAND Page at highest GTT index to DRAM
Add Vid2 to Vid1 neighbors
Can data fit in a NAND Page?
YES
Write to a NAND Page and update GTT
NO
Write to two NAND Pages and update GTT using an extended bit

Figure 8: Flow chart of add edge operation

AddEdge(V_{id1}, V_{id2}): Add edge function adds V_{id2} vertex to the neighbor list of V_{id1}. It also adds the corresponding edge weight for the added edge. As V_{id1} neighbors may be stored in a single page or may span multiple pages, we describe the operation of adding an edge for these two cases. Figure 8 shows the flowchart for the AddEdge function.

When neighbors of V_{id1} are stored in a single page: In this case, V_{id2} is added at the end of V_{id1} neighbors. The subsequent neighbors of other vertices that are stored after neighbors of V_{id1} are shifted to higher page offsets. Location pointers of these vertices that are stored at the end of the page are also updated to reflect the new location of their neighbors. This shifting of neighbors may cause an overflow in a page. In this case, when all the vertices don't fit in the page, we consider another new page and divide the vertices between the two pages such that both the pages have a roughly equal amount of empty space while maintaining the page structure described before. The reason for leaving some gaps within a page is to allow for future updates of a page without causing additional overflow.

For the newly added page, we need to index that page using the GTT entry. But if we add the GTT entry for this page in the sorted position then we may need to shift other entries in the GTT, and in the worst case, we may need to shift all the GTT entries. To avoid this scenario we keep an *extended bit* in the GTT entry. When the extended bit is set, the GTT entry doesn't point to a NAND page number but instead points to a location where GTT entries are stored contiguously for the newly added page and the old page that is split. Figure 7 shows how GTT is updated with the extended bit, when an adjacent edge is added between $V4$ and $V2$ and $P2$ page overflows for the example graph shown in Figure 6.

When neighbors of V_{id1} are stored in multiple pages: Then there are multiple GTT entries with vertex id V_{id1}. Among them, V_{id2} is added to the page pointed by the GTT entry at the highest index and after that, it is handled similarly to the above case.

An example edge addition: Here we will discuss an example of adding an edge from $V4$ to $V1$ considering the graph shown in Figure 6. First, the page containing the $V4$'s neighbor list, $P2$, is loaded into the DRAM. Then the starting offset for $V4$ neighbors is identified as 3 and $V1$ is then inserted at word 3. After insertion, as the neighbors and location pointers cannot fit in a NAND page, they are stored in two NAND pages, $P5$ and $P6$. To have roughly equal available space in $P5$ and $P6$, vertex neighbors of $V3$ are stored in $P5$, $V4$ and $V5$'s neighbors are stored in $P6$. GTT entry for $V3$ does not store the physical page number and instead it stores the location pointers where the extended GTT information is stored. And also the extended bit at $V3$'s GTT entry is set, as shown in Figure 7.

Similar to adding an edge, we also implemented adding a vertex, which inserts the vertex into GTT, and then adds multiple neighbors while maintaining the previously described page layout. Due to the space limitations we omit the description for delete and update operations on edges and vertices.

3.4 Handling Graph Updates Efficiently With Caching and Delta Graph

The graph update process described above leads to many unnecessary page writes. Since SSDs can't do in place updates, it is not possible to simply update the NAND page with new data, even if the update is as simple as just adjusting the edge weight. Each page update triggers a read-modify-write sequence for the entire page which leads to significant write amplification, in the worst case, by a factor of 1000x. For instance, a single edge weight update leads to reading the full page into a DRAM buffer, modifying the weight in the page and then writing the new 16KB page (GraphSSD page size) to a new location. To reduce the write amplification, we implemented an optimization that relies on a multi-stage update process. First, all updates are logged on the host side DRAM until sufficient number of updates have been accumulated (one page worth of updates in our current implementation) or when a timer event is triggered (a default value of 100 milliseconds is used in this work). A host side GraphSSD log manager is implemented for handling the logging functionality.

Host side logger: Certain updates such as deletions or updating an edge or vertex weight need to check whether the edge or vertex that is modified exists in the graph in the first place. Hence, the host side logger sends a request to the SSD itself to verify the presence of vertex or an edge. The logger also concurrently launches another thread to check for the edge/vertex information in the DRAM log itself (by walking the log backwards in time), since the edge/vertex being updated may still be resident in the DRAM log from a recent update request that is not yet reflected in the SSD. If such an edge/vertex does not exist either in SSD or in the DRAM the API returns a FALSE condition back to the application. Note that the request for presence check on the SSD is simply a read operation and does not trigger any page updates.

Delta graphs: When the DRAM log is full or when the timer interrupt expires the host side logger initiates a bulk update sequence. As described earlier, graph insertions may trigger a page overflow and in the worst case each insertion may trigger multiple page writes. While DRAM buffering on the host side helps with this concern, GraphSSD adopts the concept of a Delta graph [26] to further minimize the write amplification. We implement delta graphs using two vectors in SSD, namely deltaPointer and deltaUpdates. All updates for a adjacency list are appended to the deltaUpdates vector. The newly added updates for a vertex points to the previously added delta update for that vertex. DeltaPointer for that vertex points to the index in the deltaUpdates vector that contains the latest delta updates for that vertex.

Graph accesses with DRAM logs and delta graphs: The graph access mechanisms must know the presence of delta graph for a given vertex to properly reconstruct the full graph. To mark the presence of a delta graph GraphSSD sets a dirty bit in the GTL entry corresponding to that vertex. Thus when a graph access request is received, it will first access the GTL entry to identify the physical page consisting of the original graph and if a dirty bit is set in GTL entry the access mechanism then uses the DeltaPointer to reach all subsequent updates to that vertex.

Merging delta graph with the initial graph: While delta graphs allow graph updates to be gracefully handled in terms of write amplification issue, it does lead to a slow down in graph access latency. As such it is preferable to periodically merge delta graphs into the original graph. For merging the delta graph into the initial graph, we loop over GTT and identify the GTT entries whose dirty bit is set. For these vertices, we access the delta modifications and merge them with existing neighbor list data stored in the original graph. After all the vertex modifications have been merged into the graph, all the dirty bits at the GTT entries are cleared.

3.5 Consistency considerations:

Anytime there is a graph update that is logged in DRAM there is a risk of losing that state during power failures. As is the case with file buffers in OS that cache file content, during a power loss some of the data may be lost. But what is important is that a consistent view is preserved after a reboot. To create a consistent view of the graph, every update must be atomically performed. For instance, during an edge weight update, a new page needs to be created with the updated edge weight. Even in the presence of a delta graph at some point in future a page update may be initiated when merging delta graphs with the original graph. In this scenario we first create a redo-log entry before initiating the update process. The redo-log stores the GTL entry (physical page number), the deltaPointer and deltaUpdate info for the vertex that is being updated. The update process then will write the new page first, then resets the GTL dirty bit, changes GTL entry to point to the new page, and finally invalidates the old page and resets the deltaPointer entry. If there is a power failure either during the new page write process or after the new page is written but before the GTL dirty bit is reset, on a reboot the redo-log starts the entire update process by selecting another page to write (the page that was being written before the power failure will be garbage collected just like other invalid pages using the default SSD policies for handling write failures). If the power failure occurs after the GTL dirty bit is reset but before the GTL entry is updated with a new page entry the redo-log sets the GTL dirty bit back again and restarts the update process. If the power failure occurs after the GTL entry has been updated but before the old page is invalidated, the redo-log simply invalidates the old page and deltaPointers. Note that there are several optimizations that can be made to improve the performance of redo-logging. We currently focus on functionality and leave optimization for future work.

Garbage collection: We use a single bit for every page in the storage to indicate if that page is used by GraphSSD or not. If that page has been used for storing the graph data then while moving that NAND page during garbage collection, the garbage collector informs GraphSSD runtime which will in turn update the GTT entry. The page that is being moved already contains the information regarding the smallest vertex id whose neighbor lists are stored in that page. We update the GTT entry for that vertex id with the new NAND page location where the data is moved.

3.6 GraphSSD cache manager

To improve the performance for graph applications when accessing storage, graph data is cached at the host side. For algorithms based on graph data such as page rank and graph filtering, which request sequential vertex ids, there might be many requests to storage for nearby vertex data. Handling these requests to the storage adds considerable overhead and dominates the application time. To reduce this overhead we implemented a cache manager, which caches GTT on the host side, does graph command handling. GraphSSD cache manager issues requests to fetch NAND pages on a cache miss. As many vertices may reside in a NAND page, single NAND page fetch to host side cache may serve many requests on the host side itself thereby filtering requests to storage. The host side GTT is read-only and all update requests invalidate the cached GTT entry and the update is handled on the SSD itself.

During garbage collection at the SSD, data in a NAND page may be written to another NAND page. If a NAND page storing graph data is moved then new NAND page number should be updated at the host GTL cache. For updating this NAND page number on the host side cache the SSD controller automatically initiates a host cache invalidation request which is handled by the GraphSSD cache manager.

3.7 Graph command handling examples

We summarize our system implementation using two example graph access APIs; GetAdjacentVertices, and GetEdgeWeight commands. For these commands we describe how we retrieve the required data from the NAND pages.

GetAdjacentVertices(vertexID, EdgeList): Using the requested vertex id GTT is accessed to get the NAND page numbers storing its adjacent vertices. If the GTT's extended bit is set then we may have to search through the extended GTT entries to find the physical page. Once a physical page location is identified, and if the dirty bit in the GTT entry is set then it indicates that some of the neighbor information in that page has been modified. Hence, GraphSSD accesses the original graph page, and the Deltaupdates vector. Concurrently the host side logger searches the host side DRAM logs to find any cached or updated edge information for the given vertex. Finally the information from the original graph page, Deltaupdate page and the host side DRAM pages is combined to create the EdgeList buffer which is returned to the application.

Algorithm 1 Code snippet of BFS program using GraphSSD

```
/*BFS Request thread code*/
Queue.push(root)
while Queue not empty do
    top_element = Queue top element
    if top element == required element then
        Element found
        Exit
    Wait until empty slot is available in GraphSSD response queue
    Wait until empty slot is available in GraphSSD request queue
    GraphSSD.GetAdjacentVertices(top_element, EdgeList)

/*BFS Response thread code*/
Wait until EdgeList is available in GraphSSD response queue
for i=0; i < EdgeList.size(); i++ do
    if EdgeList[i] not already visited then
        Add to Queue
```

GetEdgeWeight(vertexID1, vertexID2, EdgeWeight): Using $VertexId1$ we access the GTT and fetch all the pages containing the neighbors of $VertexId1$. Using $VertexId1$ we access the NAND page containing the neighbor lists via host cache to find if $VertexId2$ is a neighbor. As briefly mentioned earlier, a separate GTT structure is used to map a vertex id to the corresponding edge weight information using the same graph layout structure as the edge connectivity information. We use the edge connectivity information to find the index location of the edge to access the edge weight. Concurrently the host side logger searches the host side DRAM logs to find any updated edge weight for the given edge. The information returned from the original graph pages is again reconciled with any updated edge weight information found in the DRAM logs to get the most recent edge weight information which is returned to the application.

4 WORKLOADS AND IMPLEMENTATION DETAILS

GraphSSD essentially provides semantic awareness to the SSDs. Instead of accessing SSDs with logical block address, it allows users to query graph related information. For this purpose, it implements basic graph access commands listed in Table 1. Users/libraries can use these basic commands to build higher-level functions. We evaluated several traditional graph applications that stress the following features 1) accessing adjacent vertices for a requested vertex, 2) accessing edge weight for a requested edge and 3) updating edge weights.

4.1 Workloads

The applications evaluated include:

BFS: BFS identifies whether a given target node is reachable from a given source node. We implemented the BFS application as shown in code snippet 1. This algorithm fetches adjacent vertices for a given vertex id repeatedly. For evaluating BFS, we select one source vertex and then varied the distance at which the destination vertex may be found at several levels along the the longest path, at 3 equal intervals from the first level to the last level.

Connected components: The number of connected components are counted in this graph application. The approach uses BFS but it performs the operations on all vertices.

Random walk: This application performs many random-walks starting from several source nodes. Starting from each source node, the application does random-walks for several iterations and in each iteration, it walks a maximum of steps. We implemented the efficient parallel random-walk algorithm described in [20]. This algorithm simulates random-walks in parallel, possibly from a large number of source vertices, and processes one vertex a time. In a step, at each vertex, all walks currently visiting that vertex are processed and moved forward. We evaluated with 100K parallel random walks, with a maximum of 10 steps from the source and considered several iteration values, 10, 50, 100, and 1000.

Maximal independent set: We implemented the maximal independent set algorithm as described in [35]. It is an iterative algorithm based on Luby's classic parallel algorithm[24].

Page rank: [33] The page rank is a classic graph update algorithm and our implementation sends edge updates if they are greater than a certain threshold (0.4). We set the maximum number of iterations to 5.

Graph Update benchmark: Since GraphSSD provides significant support for graph updates, we also implemented a graph update kernel that adds edge and vertex information. The updates are maintained as delta graphs and are finally merged into the initial graph.We intersperse the graph updates with a total of 1000 get adjacent queries on vertices selected using the latest read model. In the latest read model, the newly added updates are accessed the most. Latest read model represents the widely used news feed, social media, where newly posted data is accessed the most [6]. We consider 95% of the get adjacent queries on vertices that are being updated or newly added to the graph.

Non-intrusive NVMe Implementation: All the GraphSSD APIs are implemented by extending existing NVMe read/write commands. We used the unused bytes in read and write NVMe commands to specify the GraphSSD commands. We used these unused bytes to pass the vertex id, end vertices of an edge and an opcode encoding to indicate the API operation being requested.

4.2 Baseline system

For the baseline system, we considered normal SSDs where the graph is stored in CSR format (described in 2.2). The baseline uses block based access to reach the rowPtr, colIdx and val vectors. Each access to the rowPtr, colIdx or Val vector is first translated to a physical page number using traditional FTL. The baseline system also uses file caching on the host to cache multiple pages; the size of the host side file cache is 1GB in our implementation. As we show later in our results section, host side caching is critical for implementing a robust baseline that can eliminate many NAND page accesses.

We also compare results with the popular out-of-core GraphChi framework. While comparing with GraphChi we use the same host side memory cache size as GraphSSD. For all the applications, when executing over GraphSSD, Baseline, and GraphChi, application data other than the graph data, such as visited vector in BFS application, value vector in page rank application, etc. are allocated in main memory.

4.3 Caching and Multi-threading

We implemented host side caching using LRU policy for both GraphSSD and baseline. Doubly linked list and hashmap are used to efficiently implement the LRU policy. To simulate out-of-core graph algorithms, we consider host cache size of 1GB as the default size.

To generate I/O request parallelism our baseline and GraphSSD implementations both provide a non-blocking request interface to graph application threads. To support non-blocking calls, we implement a request queue and response queues. In the request queue, graph data requests are posted from the application. Graph data responses to those requests are pushed into the response queues. The cache manager is also parallelized using multi-threading to maximize the throughput, and locks were sparingly used for synchronizing between the threads as necessary.

From storage, GraphSSD loads NAND page granularity chunks into the host cache as location pointers are stored at the end of the NAND page and vertex data is stored at the start of the NAND page. In our baseline we also load NAND page granularity chunks into the file buffer host cache.

5 EVALUATION

We evaluated GraphSSD using the open-source SSD (OpenSSD) development platform [31, 36]. The OpenSSD development platform equips Xilinx Zynq-7000 programmable SoC that embeds a dual-core ARM Cortex-A9 processor [39]. Hence the FPGA-based programmable chip works as an SSD controller SoC on the SSD platform. PCIe interface and NAND flash channels are implemented as hardware logic on the programmable gate arrays, and the embedded ARM core runs the SSD firmware implementing the command handling, page buffer management and FTL functions. We implemented GraphSSD on the existing SSD firmware modifying the FTL part, command handling and host side library which manages cache on the host side. The OpenSSD platform encloses 1 GB DDR DRAM and 2 TB Hynix H27Q1T8YEB9R NAND flash DIMMs connected to the programmable SoC. The SSD board communicates with the host system via the PCIe Gen2$\times$8 interface, which supports up to 4 GB/s bandwidth. NAND page size in OpenSSD platform is 16KB. Host system uses a logical sector of size 4KB.

We configured the host system with Intel i7-4790 CPU running at 4 GHz and 16 GB DDR3 DRAM. In order to extend NVMe commands for GraphSSD, the NVMe host driver on Linux Kernel version 3.19 was enhanced.

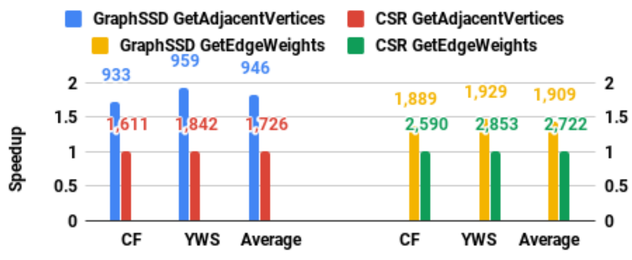


Figure 9: Relative performance of basic GraphSSD API

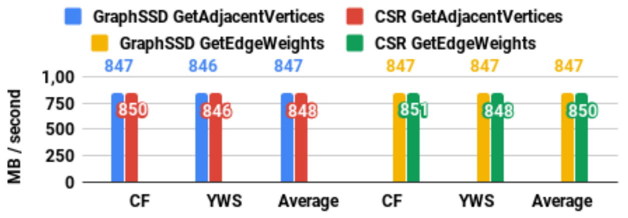


Figure 10: Bandwidth for basic GraphSSD API (MB/second)

Graph dataset: To evaluate the performance of GraphSSD, we selected two real-world datasets, one from the popular SNAP dataset [23] called *com-friendster* graph, and another is a popular webgraph from Yahoo Webscope dataset [40]. These graphs are all undirected graphs and for an edge, each of its end vertices appears in the neighboring list of the other end vertex. The datasets are listed in Table 2. For updates, we used a real-world YouTube dataset [29]. The graph has 1M vertices, 4.4M edges at the start. During the update process 38K new vertices and 550K new edges are added to the graph. Based on the timestamps in the dataset we divided these updates into 10 equal snapshots to test delta graph generation and merging functionality.

Dataset name	Number of vertices	Number of edges
com-friendster (CF)	124,836,180	3,612,134,270
YahooWebScope (YWS)	1,413,511,394	12,869,122,070

Table 2: Graph dataset

6 EXPERIMENTAL RESULTS

6.1 Performance of basic APIs

We first present performance results of the basic graph access APIs that are provided by GraphSSD, namely GetAdjacentVertices and GetEdgeWeight APIs, before presenting the application-level performance. We ran each command 1 million times using a random vertex or edge as the starting point for the query.

GetAdjacentVertices API: The left bars in Figure 9 shows the relative performance of GraphSSD over baseline for the *GetAdjacentVertices* API. GraphSSD outperforms baseline by 1.85×. There are two potential sources for performance improvement with GraphSSD. First, the baseline system accesses the *rowPtr* and *colIdx* vectors using block interfaces. GraphSSD on the other hand uses GTL to find the adjacent vertex list for a given vertex. When accessing SSD the baseline has to access two different NAND pages for the *rowPtr* and *colIdx* vectors but GraphSSD uses semantic knowledge through GTL to reduce NAND page access count. The number above each bar in the graph shows the number of NAND pages accessed to the nearest thousand. Even though there were million queries to random vertices, host side caching helps in reducing the number of NAND pages accesses. The performance of GraphSSD when compared to the baseline is not doubled, as *rowPtr* vector is compact when compared to *colIdx* vector, and baseline system caches that vector on the host side effectively to reduce the *rowPtr* related NAND page accesses.

We also considered the performance degradation in the baseline due to the serialization bottleneck. The baseline has to first access the *rowPtr* to find the index to access the *colIdx*. Hence, the two accesses may be sequentialized. Such a serialization will essentially reduce the SSD request rates which will result in lower data bandwidth between SSD and the host. However, the delay due to the sequential nature of the two accesses can be hidden effectively if multiple parallel requests can be issued; then the *rowPtr* access and *colIdx* vectors of different requests can be interleaved. In fact as we described earlier, our baseline is highly multithreaded and is able to issue multiple read requests. Since we tested the *GetAdjacentVertices* API with million randomly selected vertex queries these queries can be issued in massively parallel manner, which in fact reduces the serialization bottleneck. This fact can be easily verified by looking at the bandwidth utilization between SSD and the host as shown in Figure 10. The Y-axis shows the total number of MBs transferred per second between the SSD and the host for both the baseline and GraphSSD. Both approaches transfer roughly the same amount of data per second. Hence, the serialization bottleneck is not a critical performance limiter.

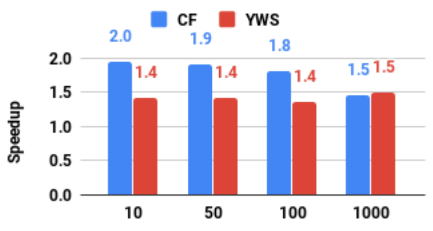
GetEdgeWeight API: The right side bars in Figure 9 compares the performance of loading the weight for an edge using the *GetEdgeWeight* API. GraphSSD outperforms the baseline by 1.42×. Unlike accessing the adjacent vertex lists which may span many pages for densely connected vertices, accessing an edge weight requires accessing at most one more NAND page in the baseline where the weight is stored (that too when there is a host cache miss). Hence, the performance gap is narrower. Again, the parallelization of queries reduces any serialization bottleneck as seen from the bandwidth utilization graphs in Figure 10 (right three sets of bars).

6.2 Application performance

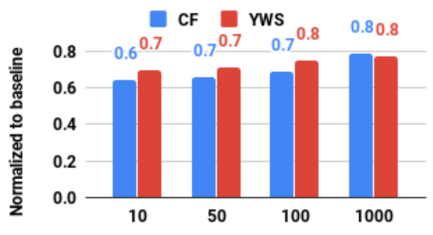
We first use Random-walk application to present detailed results and analysis followed by the performance results for all applications.

Random-walk: Figure 11a plots the performance (Y-axis) of Random-walk on GraphSSD normalized over the baseline system. X-axis shows the number of iterations of random-walks done starting from each source node. The performance improvement of GraphSSD on average is about 1.6× better when compared to the baseline system over a wide range of number of random-walks from a source node.

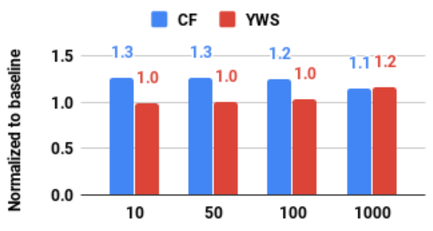
In case of the Random-walk application, GraphSSD's performance improvements in fact do come from the two reasons we discussed earlier: NAND page access counts and bandwidth utilization. In this application as the next vertex in the walk is visited randomly, baseline is less effective in caching the *rowPtr*. Using a compact representation of GTL GraphSSD can reduce the number of NAND page accesses. This observation is quantified in Figure



(a) GraphSSD Speedup

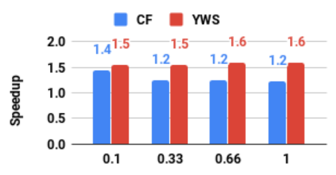


(b) NAND page access counts

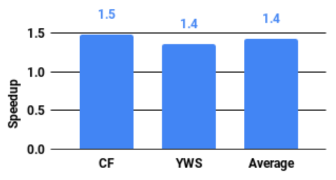


(c) SSD bandwidth utilization

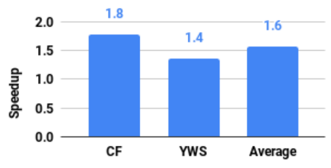
Figure 11: Random-walk performance relative to baseline (X-axis is the number of random-walks performed)



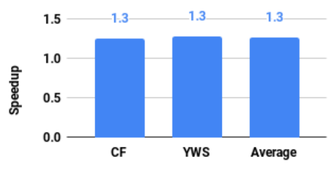
(a) BFS (X-axis is traversal depth)



(b) Connected components



(c) Maximal independent set



(d) Page rank

Figure 12: Application speedup relative to baseline

11b, which shows the NAND page accesses relative to baseline. Performance benefits also come due to the better bandwidth utilization of GraphSSD relative to the baseline. In the baseline, serialization problem of accessing the *rowPtr* first before accessing the *colIdx* causes underutilization of SSD bandwidth. But GraphSSD is able to better cache GTL and access the adjacent vertex information in parallel leading to an improved bandwidth utilization as seen from Figure 11c.

All other applications: Figure 12 shows the performance improvements for the other applications on GraphSSD over the baseline system. For BFS, we breakdown the performance along the X-axis based on the fraction of the total number of levels in the graph that must be traversed to find a connection from the source node to the destination node. For instance, a value of 0.1 on X-axis means the destination node was found after traversing one tenth of the longest path in a graph. We generated queries at four different depths, 0.1, 0.33, 0.66 and 1 and the results are plotted in the figure. When compared to the baseline system, GraphSSD improves performance on average by 1.40×, 1.42×, 1.56× , and 1.29×, for bfs, connected components, maximal independent set, and page rank applications, respectively. Just as we discussed earlier, the reason for the improved GraphSSD performance for these applications is a combination of the fewer NAND page accesses and more effective utilization of bandwidth, due to space constraints we omit showing that data for these applications.

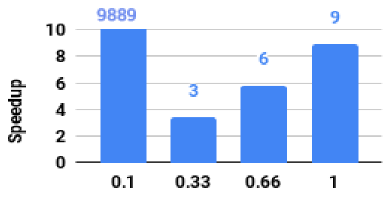
6.3 Comparison with GraphChi

Figure 13 compares the performance of GraphSSD over GraphChi for two benchmarks BFS and Page Rank. GraphChi is designed for vertex centric programming model and applications such as BFS and connected components are ill-suited for this programming model. GraphChi is a vertex-centric programming framework and in most of the iterations, all the shards (partitioned graph data) are fetched from the storage repeatedly and only a few active vertices are actually used in the computation. Hence, even though we collected results, it will be unfair to GraphChi to compare its performs when the applications are not suited for the programming model it supports. Hence, the only purpose of us plotting the GraphChi comparison results for BFS is to demonstrate that GraphSSD is more versatile as it supports broader set of programming models. The first two graphs in the Figure 13 show the significant performance degradation with GraphChi for the two graph datasets that were used in the evaluation. Clearly, accessing all the vertices in each iteration as required by vertex centric programming model is not well suited for BFS. As the fraction of traversed graph increases, the performance of GraphChi worsens, since many iterations may be necessary for the vertex centric programming model to converge. And during each iteration all the graph shards may be read by GraphChi.

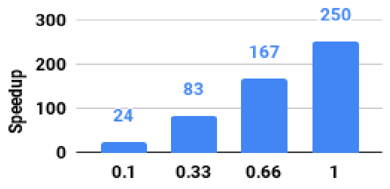
On the other hand, applications such as PageRank are better suited for vertex centric programming models where many vertices may be traversed repeatedly. Figure 13c, compares PageRank application on GraphSSD over the GraphChi. GraphSSD still outperforms GraphChi by 2.62× even for PageRank. Even though PageRank updates many vertices, *not all* vertices are active even in PageRank in each iteration. The sharding based data structure in GraphChi requires almost all shards to be fetched for each iteration into memory even though the number of compute operations per shard are quite small in some shards. The variations seen across different graphs are simply a function of the graph structure and edge weights which influence the PageRank convergence process.

6.4 GraphSSD Overheads

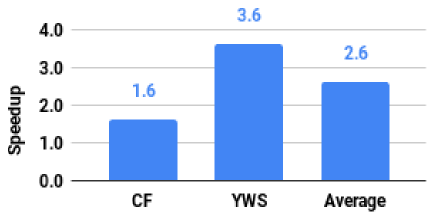
With the GraphSSD data layout, GTT size for the *Com-Friendster* and *YahooWebGraph* is 8M and 34MB, respectively. These GTT structures are quite small compared to the typical DRAM size in SSDs. The reason for the small size of GTT is the compact layout we propose for GraphSSD where multiple vertices that fit in a single page have a single GTT entry. Packing multiple vertices into one GTT entry is very common in both the graphs we studied; only rarely a vertex requires multiple GTT entries (≈0% in CF and 0.01% in



(a) BFS relative to GraphChi for CF dataset



(b) BFS relative to GraphChi for YWS dataset



(c) Page rank relative to GraphChi

Figure 13: Speedups relative to GraphChi

YWS). Since all these graphs are dominated by sparsely connected vertices the compact layout of GrahSSD is another key element for boosting performance. As such the size of the GTT is small. If one were to use one entry per each vertex the size of the GTT would be atleast 1GB and 11.2GB, respectively for the two graphs studied in this work.

Space overhead: While packing the adjacent vertices into a NAND page, for the simplicity of design, we used new NAND page if the adjacent vertices doesn't fit in the existing free space (described in subsection 3.2). The space overhead due to this design choice is 4.8% and 4.9% for the CF and YWS graphs, respectively. This space overhead can be easily avoided with a more complex design that tracks a vertex neighbors split across pages.

6.5 Graph updates

Figure 14a shows the performance of accessing graphs that are continually updated as described in the graph update benchmark earlier. The X-axis shows the number of delta graphs that are chained. The performance of GetAdjacentVertices degrades slowly with increasing number of updates which are stored as delta graph chains. Recall that accessing the graph after multiple updates may require traversing the UpdatePointer chains which slows down the performance.

Figure 14b shows the latency of the merge process with varying degrees of empty slots available in the page to prevent page spills. With a 10% empty space in each page, the merge process takes 2.4 seconds in our graph benchmark, and the time increases to 2.7 seconds when the empty space per page is limited to 5% which leads to page splits when a new edge or vertex is being added. The time increases to 3.9 seconds when the empty slots are just 1% of the total page size.

Since the merge process itself is slow, any access during the merge process itself may see variable latency depending on whether the accessed vertex is already merged in which case it is a normal page access, or if it is not merged at all in which case the access may need to go through a chain of pointers. Figure 14c shows the variability in access latency on X-axis averaged over 1000 random GetAdjacentVertices queries compared to a graph that has no updates (or has been fully merged already). The X-axis shows the fraction of the graph updates that are already merged in 10 equal intervals of 10% each. The latency penalty decreases steadily as the merge process progresses. Further, the performance with different levels of empty space availability in a page is almost the same, even though they lead to different page splits, as we use random GetAdjacentVertices queries and there is less caching benefit.

During the merge process in GraphSSD, one only needs to merge NAND pages that have any updates. If any of the updated NAND pages overflow then empty NAND pages can be used to store these updates (described in subsection 3.3). For graphs where updates are in fewer NAND pages, this helps us in faster merging of the updates when compared to CSR baseline. In CSR baseline the entire graph structure needs to be written to another place [26]. Reducing the number of NAND pages written helps in reducing the critical wear out of NAND pages in SSD, as they can only support a limited number of writes during their lifetime.

7 RELATED WORK

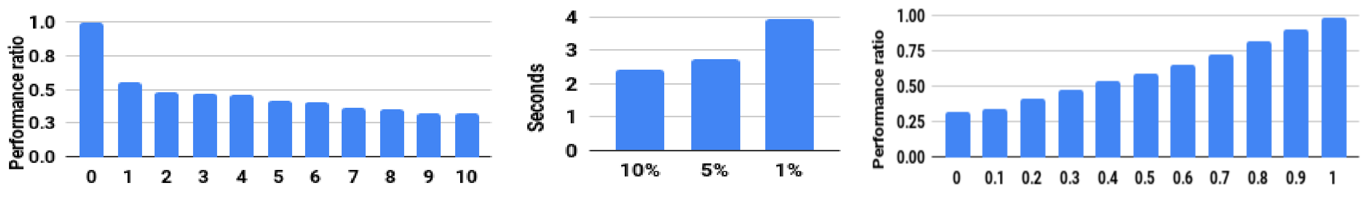
7.1 Storage system

Exploiting the computation power of the SSD controller SoC can be an opportunity for offloading the computation burden and curtailing the data traffic in SSD for data-intensive applications. *Active disk* research proposed the possibility of in-storage processing for data-intensive applications with data processor on the storage disk platform [1, 34]. A plethora of complex in-storage processing applications have been proposed as SSD controller embeds more powerful application processors to handle massive data parallelism from multi-channel NAND flash memory [3, 5].

Summarizer [19], Active Flash [3, 37, 38], SmartSSD [7, 17], Active Disks Meets Flash [5] and Biscuit [9] try to utilize the embedded cores in a modern SSD to reduce the redundant data movement to free-up the host CPUs and main memory. Active Flash [37] for instance presents an analytic model for evaluating the potential for in-SSD computation. Summarizer [19] presents a detailed description of the application development environment to enable offloading work to SSD controller. SmartSSD [7] focuses on how to improve specific database operations, such as aggregation, using in-SSD computation. Biscuit [9] uses a flow-based programming model to enable code offloading to SSDs. SSD's data processing ability has also been exploited to support key-value interface for SSD. KAML proposed key-value interface for SSDs instead of the traditional block-based I/O interface [14]. To efficiently support key-value storage they use hash-based key-value mapping tables. To the best of our knowledge, GraphSSD is the first investigation to employ the graph semantic aware mapping structure in an SSD.

7.2 Graph processing

Researchers have focused on enhancing the performance of graph applications on the hardware and software front. Hardware approaches exploit the graph processing acceleration engines near memory since



(a) Performance with delta snapshots **(b) Latency of merge process** **(c) Performance during merge process**

Figure 14: Graph update performance

large graph structure requires huge data transfer from main memory. Ahn et al. proposed Tesseract hardware accelerator that uses in-memory processing to improve memory bandwidth utilization [2]. Graphicionado is a graph-specific hardware accelerator which utilizes vertex programming model observed in wide range of graph applications [10]. These works optimize the memory system to guarantee data parallelism for vertex traversals. While the prior graph hardware accelerators focus on graphs that fit in main memory, GraphSSD tackles the data I/O bottleneck of large graph data structure on storage devices. ExtraV does hardware graph acceleration in front of storage to reduce the burden of graph management on the host processor and also support processing intensive compression to reduce the storage accesses [22]. However they use the SSD as a block device and is orthogonal to our work. Such hardware acceleration approaches in the front end of storage can also be employed with GraphSSD to further improve performance.

On the software front, on a single node system, GraphChi [21] and PartitionedVC [28] are out-of-memory systems which try to efficiently support vertex-centric graph programming. GraphChi proposes a sharding based graph format to reduce random I/O accesses from storage. PartitionedVC improves over GraphChi to access storage based on the number of active vertices/edges in a vertex-centric superstep. TurboGraph is an external-memory graph engine targeting graph algorithms expressed in sparse matrix-vector multiplication [12]. However, it is difficult to implement graph applications such as triangle counting on such a framework. They use large page sizes in multiple of MBs making it inefficient to read adjacent vertices selectively. FlashGraph implements graph engine on top of SSD file system to maximize parallel execution [41]. In order to hide data transfer overhead, FlashGraph also overlaps graph computation and data I/O from SSDs where only edge lists are stored. Mosaic [25] accelerates out-of-core graph processing on heterogeneous machines. All the existing out-of-core memory systems only look at flash as a block device. GraphSSD on the otherhand makes SSD aware of graph semantics, and proposes a layout scheme that takes SSD organization in to consideration. By customizing a widely used compressed graph representation we reduce the number of costly NAND page accesses to access graph data in out-of-core graph processing and support efficient merging of graph updates which hitherto was a problem while using the compressed graph representation. CSR format based graph processing works show that it is efficient for out-of-core graph processing, and performs better when compared to other out-of-core graph formats such as GraphChi format [15, 16, 22, 26, 30]. In this work, we consider a CSR format based graph framework and GraphChi as a baseline and show significant performance improvements.

GraFBoost is a vertex-centric programming model for out-of-memory graph analytics that reduces the overhead of random updates by using a log structure [15]. This work is orthogonal to our work of reducing the random access to the graph data by a NAND page organization aware graph layout. GraFBoost can be potentially incorporated within GraphSSD to further improve the performance.

There have been several works to support dynamically changing graphs. Kineographs, Chronos, and Stinger are in-memory graph processing frameworks for handling dynamic graphs [4, 8, 11]. GraphSSD handles dynamic graphs that go beyond in-memory graphs. LLAMA proposes efficient support for whole-graph analysis on consistent views of data while supporting streaming graph updates [26]. It proposes to store the streaming graph updates in snapshots along side the initial graph and merge a batch of snapshots to amortize the costly merge operation, limit the space utilization of snapshots and reduce the performance penalty while accessing from the snapshots containing the latest graph updates. GraphSSD's delta graphs are inspired by the LLAMA approach.

8 CONCLUSION

Graph applications access increasingly large graphs that are hobbled by storage access latency. In this work, we proposed GraphSSD, a graph-semantic-aware SSD framework that allows storage controllers to directly access graph data natively on the flash memory. We presented graph translation layer (GTL), which translates the vertex ids to physical page address on the flash memory media directly. In conjunction with GTL, we propose an efficient indexing format that reduces the overhead of GTL with only a small increase in the per-page metadata overhead. We also presented multiple optimizations to handle graph updates using delta graphs which are merged to reduce update penalty while at the same time balancing the write amplification concerns. We implemented GraphSSD framework on an SSD development platform to show the performance improvements over two different baselines. Our evaluation results show that the GraphSSD framework improves the performance by up to 1.85× for the basic graph data fetch functions and on average 1.40×, 1.42×, 1.60×, 1.56×, and 1.29× for the widely used breadth-first search(BFS), connected components, random-walk, maximal independent set, and page rank applications, respectively.

9 ACKNOWLEDGMENT

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0053, NSF grant 1719074, Samsung award 079856. Tseng is supported by NSF grants 1657039 and 1812987. Koo is supported by the National Research Foundation (NRF) grant funded by the Korea government (MSIT) (No. NRF-2018R1C1B5086594) and Institute of Information & Communication Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-00533, Research on CPU vulnerability detection and validation). The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense, the U.S. Government, or the Korean government.

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '98, pages 81–91, New York, NY, USA, 1998. ACM.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 105–117, New York, NY, USA, 2015. ACM.

[3] Simona Boboila, Youngjae Kim, Sudharshan S. Vazhkudai, Peter Desnoyers, and Galen M. Shipman. Active Flash: Out-of-core data analytics on flash storage. In *IEEE 28th Symposium on Mass Storage Systems and Technologies*, MSST '12, pages 1–12, April 2012.

[4] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the Pulse of a Fast-changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. ACM.

[5] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active Disk Meets Flash: A Case for Intelligent SSDs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 91–102, New York, NY, USA, 2013. ACM.

[6] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[7] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[8] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5, Sept 2012.

[9] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A Framework for Near-data Processing of Big Data Workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 153–165, Piscataway, NJ, USA, 2016. IEEE Press.

[10] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '16, pages 1–13, Oct 2016.

[11] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM.

[12] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 77–85, New York, NY, USA, 2013. ACM.

[13] Amber Huffman. NVM Express, 2013.

[14] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. Kaml: A flexible, high-performance key-value ssd. In *HPCA*, pages 373–384. IEEE, 2017.

[15] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. GraFBoost: Accelerated Flash Storage for External Graph Analytics. *ISCA*, 2018.

[16] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. BigSparse: High-performance external graph analytics. *arXiv preprint arXiv:1710.07736*, 2017.

[17] Yangwook Kang, Yang suk Kee, Ethan L. Miller, and Chanik Park. Enabling cost-effective data processing with smart SSD. In *IEEE 29th Symposium on Mass Storage Systems and Technologies*, MSST '14, pages 1–12, May 2013.

[18] Jin-Yong Kim, Sang-Hoon Park, Hyeokjun Seo, Ki-Whan Song, Sungroh Yoon, and Eui-Young Chung. NAND Flash Memory With Multiple Page Sizes for High-Performance Storage Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):764–768, Feb 2016.

[19] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading Communication with Computing Near Storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 219–231, New York, NY, USA, 2017. ACM.

[20] Aapo Kyrola. Drunkardmob: billions of random walks on just a pc. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 257–264. ACM, 2013.

[21] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[22] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H Peter Hofstee, Gi-Joon Nam, Mark R Nutter, and Damir Jamsek. ExtraV: boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment*, 10(12):1706–1717, 2017.

[23] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection.

[24] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[25] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 527–543. ACM, 2017.

[26] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 363–374. IEEE, 2015.

[27] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[28] Kiran Kumar Matam, Hanieh Hashemi, and Murali Annavaram. PartitionedVC: Partitioned External Memory Graph Analytics Framework for SSDs. *arXiv e-prints*, page arXiv:1905.04264, May 2019.

[29] Alan Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, Department of Computer Science, May 2009.

[30] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. GraphBIG: understanding graph computing in the context of industrial solutions. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[31] OpenSSD. Open-Source Solid-State Drive Project for Research and Education. http://openssd.io.

[32] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy Efficient Architecture for Graph Analytics Accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 166–177, Piscataway, NJ, USA, 2016. IEEE Press.

[33] Pagerank application,. https://github.com/GraphChi/graphchi-cpp/blob/master/example_apps/streaming_pagerank.cpp.

[34] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active Disks for Large-Scale Data Processing. *Computer*, 34(6):68–74, June 2001.

[35] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7):577–588, 2014.

[36] Yong Ho Song. Cosmos+ OpenSSD: A NVMe-based Open Source SSD Platform. In *Flash Memory Summit 2016*, Santa Clara, CA, USA, 2016.

[37] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. Active Flash: Towards Energy-efficient, In-situ Data Analytics on Extreme-scale Machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 119–132, Berkeley, CA, USA, 2013. USENIX Association.

[38] Devesh Tiwari, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter J. Desnoyers. Reducing Data Movement Costs Using Energy Efficient, Active Computation on SSD. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower '12, Berkeley, CA, USA, 2012. USENIX Association.

[39] Xilinx. Zynq-7000 All Programmable SoC Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[40] Yahoo WebScope. Yahoo! altavista web page hyperlink connectivity graph, circa 2002. http://webscope.sandbox.yahoo.com/, 2018.

[41] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, FAST '15, pages 45–58, Santa Clara, CA, 2015.