Blaze: Fast Graph Processing on Fast SSDs

Juno Kim
Computer Science and Engineering
University of California, San Diego
La Jolla, U.S.
juno@eng.ucsd.edu

Steven Swanson
Computer Science and Engineering
University of California, San Diego
La Jolla, U.S.
swanson@eng.ucsd.edu

***Abstract*—Out-of-core graph processing is an attractive solution for processing very large graphs that do not fit in the memory of a single machine. The new class of ultra-low-latency SSDs should expand the impact and utility of out-of-core graph processing systems. However, current out-of-core systems cannot fully leverage the high IOPS these devices can deliver.**

We introduce Blaze, a new out-of-core graph processing system optimized for ultra-low-latency SSDs. Blaze offers high-performance out-of-core graph analytics by constantly saturating these fast SSDs with a new scatter-gather technique called *online binning* that allows value propagation among graph vertices without atomic synchronization. Blaze offers succinct APIs to allow programmers to write efficient out-of-core graph algorithms without the burden to manage complex IO executions. Our evaluation shows that Blaze outperforms current out-of-core systems by a wide margin on seven datasets and a set of representative graph queries on Intel Optane SSD.

***Index Terms*—Network theory (graphs), Data analysis, High performance computing, Parallel processing**

I. Introduction

Out-of-core graph processing enables the processing of large graphs that do not fit in the available main memory of a single machine by judiciously moving data between memory and storage. The design of out-of-core graph processing systems has evolved for nearly a decade [15], [20], [11], [29], [27], [16], [12] with a strong focus on optimizing IO performance to minimize the overhead of slow storage access. With significant improvements in storage technology, the design of these systems has also been tailored to benefit from the improved performance of more advanced devices. Well-optimized, out-of-core graph processing systems have shown that they provide attractive performance with lower cost and complexity compared to the complex distributed graph processing solutions that spread the graph in the memories of multiple machines [10], [13], [17], [25].

The design of out-of-core graph processing systems now faces new challenges and opportunities as more performant storage technologies emerge. For example, modern SSDs like Intel Optane SSD or Samsung's Z-NAND offer an order-of-magnitude higher bandwidth compared to conventional SSDs. The most critical aspect of these new devices is their improved bandwidth and their symmetric performance between sequential and random IO. We refer to these modern SSDs as Fast NVMe Drives (FNDs), the same terminology used by previous literature [14].

In this work, we present Blaze, an open-source, out-of-core graph processing system optimized for FNDs.[1] Specifically, Blaze aims to keep the FNDs constantly saturated to achieve high performance. For this, Blaze introduces a novel scatter-gather scheme called *online binning* that propagates values among graph vertices without the synchronization overhead while achieving good load balance, the goal previous techniques like synchronization and message passing cannot achieve simultaneously. For balanced IO, Blaze uses page-interleaved Compressed-Sparse Row (CSR) format which helps increase IO utilization from multiple SSDs while minimizing IO amplification. Finally, Blaze allows the programming of efficient out-of-core graph algorithms under the well-known API, EDGEMAP and VERTEXMAP, first introduced in Ligra [22]. We extend them to be efficiently used in out-of-core graph processing.

We evaluate Blaze against two state-of-the-art, open-sourced, out-of-core graph processing systems, FlashGraph [27] and Graphene [16]. These systems are also designed for random IO unlike early-generation out-of-core graph processing systems [15], [20], [29]. Compared to FlashGraph and Graphene, Blaze offers substantial speedups (up to 13.6×) in a wide variety of workloads as we describe in Section V.

Overall, we make the following contributions in this paper.

- An analysis of two recent out-of-core systems, FlashGraph [27] and Graphene [16], revealing

[1]The code is available at https://github.com/NVSL/blaze.

their performance problems on FNDs
- A novel atomic-free, scatter-gather scheme called *online binning* that applies graph algorithms on disk-resident graphs with low CPU overhead and high load balance
- An extension of EDGEMAP API to the out-of-core graph processing stack
- An open-sourced implementation of Blaze

This paper is organized as follows. Section II describes the background and motivation of this work. Section III discusses the root cause of the low performance of existing systems on FNDs. Section IV describes the design and implementation of Blaze. Section V describes the experimental setup and results. Section VI discusses related works and Section VII concludes.

II. Background and Motivation

In this section, we describe the background on out-of-core graph processing and the performance characteristics of modern FNDs. Then we further discuss the performance problems of current out-of-core graph processing systems when they run on FNDs, motivating our work.

A. Out-of-core Graph Processing

Out-of-core graph processing enables the processing of very large graphs that do not fit in the main memory of a single machine – by placing the graph on the secondary storage and conducting frequent IO to process graph algorithms on storage-resident graphs. Compared to the distributed graph processing that places graphs on an aggregated memory of multiple machines, out-of-core processing offers similar or better performance without the need to deal with the complexity of distributed computing [15], [20], [29], [27], [16], [18].

Current systems The design of out-of-core graph processing systems have evolved in step with advances in storage performance. For instance, out-of-core systems designed around the early 2010s were optimized for sequential disks [15], [20]. To maximize the IO performance, these systems access disk-resident graphs *sequentially* at the cost of accessing more data than necessary. Even with this potential access amplification, they benefit from sequential access due to the significant performance gap between sequential and random access on early-generation disks.

However, more recent systems [27], [16] make a different tradeoff since storage devices started offering fast random access with little performance gap with sequential access. These systems do not place a high priority on issuing large sequential IO, achieving lower IO amplification than prior systems. Blaze follows the same principle but further optimizes for FNDs to maximize the benefit of fast random IO that existing systems fail to leverage.

SSD	Model	Seq. 4 kB read	Rand. 4 kB read
NAND	Intel SSD DC S3520 (2016)	386 MB/s	132 MB/s
Optane	Intel Optane SSD DC P4800X (2017)	2550 MB/s	2360 MB/s
Z-NAND	Samsung 983 ZET (2018)	3400 MB/s	3072 MB/s
V-NAND	Samsung 980 Pro (2020)	3500 MB/s	2827 MB/s

TABLE I: **The evolution of storage bandwidth.**

Processing models Out-of-core graph processing systems are classified into two models based on where they keep the vertex data. The first model is the fully-external model where the vertex data is kept on storage along with the edges. The second model is the semi-external model where the vertex data is kept fully in DRAM.

The choice between two models is determined by the available memory budget for a machine and the size of target graphs. Performance-wise, the systems with the semi-external model often outperform the ones with fully-external model as less IO is required for the former.

Blaze adopts the semi-external model for better performance while minimizing the use of DRAM.

B. Evolution of Storage Performance

Recent advances in storage technology present new challenges and opportunities for the design of out-of-core graph processing. The most notable performance trend in storage device is *symmetric high bandwidth*. For instance, Intel Optane SSD [1] achieves about 2.5 GB/s of read bandwidth for both sequential and random 4 kB access while Samsung's Z-NAND [3] and V-NAND SSD [2] show similar performance characteristics (Table I).

We confirm this by profiling the read bandwidth of the first two SSDs, NAND SSD and Optane SSD, shown in Table I. [2] The result confirms the aforementioned performance trend – On NAND SSD, random 4 kB read performs only 34% of sequential read bandwidth, showing a large performance gap between random and sequential access. However, the gap is only within 10% on Optane SSD. Also, the absolute bandwidth has undergone a significant improvement – Optane SSD shows 6.6× and 17.9× higher bandwidth than NAND SSD in sequential and random read, respectively. In summary, the

[2]In this work, we do not focus on the write performance of SSDs as our target workloads do not incur writes to the underlying storage.

Dataset	Short	|V|	|E|	Distribution	Diameter	Type
rmat27	r2	134	2147	power	10	synthetic
rmat30	r3	1074	17180	power	11	synthetic
uran27	ur	134	2147	uniform	10	synthetic
twitter	tw	61	1468	power	75	real
sk2005	sk	51	1949	power	205	real
friendster	fr	124	1806	power	56	real
hyperlink14	hy	1727	64422	power	790	real

TABLE II: **Target graphs.** The number of vertices (|V|) and edges (|E|) is in millions. "Short" denotes short names for datasets.

huge improvement in storage performance opens new challenges and opportunities in leveraging fast random IO in out-of-core graph processing.

C. Target Datasets

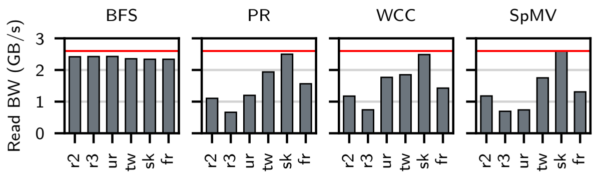
Table II shows our target datasets throughout this paper. We chose these input graphs as they are topologically diverse and different in size. The *rmat27*, *rmat30*, and *uran27* graphs are synthetic while *twitter*, *sk2005*, *friendster*, *hyperlink2014* are from real-world. Six graphs except uran27 follow a power-law degree distribution while uran27 follows a normal degree distribution. The uran27 is the most adversarial graph [5] as it has *no locality* – there are no popular vertices (no temporal locality) and neighbors are not close to each other (no spatial locality), so it well represents the other extreme in our choice of input graphs.

D. Issues with Current Out-of-core Systems

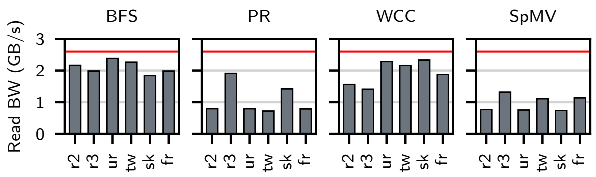
Current out-of-core graph systems cannot efficiently utilize the FND's bandwidth. FlashGraph [27] and Graphene [16], two recent out-of-core graph processing systems optimized for random IO, illustrate this problem – they fail to utilize the high throughput of FNDs, leading to a suboptimal performance on representative workloads.

We confirm this by measuring the average IO bandwidth utilization of both systems on an Intel Optane SSD with various graph workloads (Figure 1). We used six graph inputs (r2, r3, ur, tw, sk, fr) from Table II and ran five queries – Breadth-First Search (BFS), PageRank (PR), Weakly-Connected Components (WCC), Sparse Matrix Vector Multiplication (SpMV), and Betweenness Centrality (BC). For all measurements, we used 16 threads for a fair comparison.

In both FlashGraph and Graphene, IO utilization significantly varies by input graph and query. Both systems achieve high IO bandwidth regardless of the input graph for BFS. However, for PR, WCC, SpMV – more complex queries than BFS – both systems show low IO bandwidth depending on the underlying graph. In the worst case, FlashGraph achieves only 23% of the device bandwidth for PageRank on rmat30 graph while Graphene



(a) FlashGraph



(b) Graphene

Fig. 1: **Underutilized IO in FlashGraph and Graphene.** Red line: the maximum read bandwidth of Optane SSD.

Systems	Skewed computation	Skewed IO	Fast IO & slow computation
FlashGraph [27]	Yes	No	No
Graphene [16]	No	Yes	Yes
Blaze	No	No	No

TABLE III: **System comparison.** Blaze avoids the root causes of low IO utilization on FNDs.

achieves 30% of it for PageRank and SpMV on various graphs.

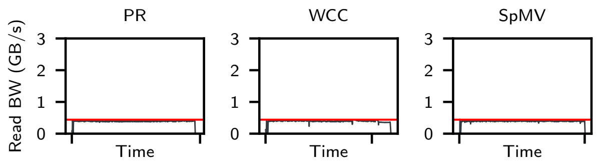
In the following section, we investigate the root cause of why FlashGraph and Graphene suffer such low IO utilization on FNDs.

III. Reasons of Low IO Utilization in Current Systems

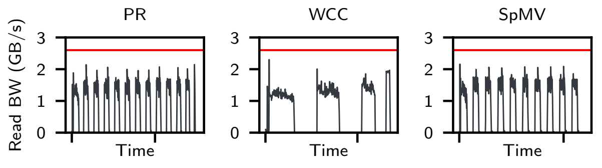
The root cause of low IO utilization in FlashGraph and Graphene on FNDs are *skewed computation*, *skewed IO*, and *fast IO, slow computation*. We elaborate on each case in more detail.

A. Skewed Computation

Parallel graph processing requires synchronization to avoid data races when updating the algorithm-specific vertex data concurrently with multiple threads [19], [22]. However, synchronization primitives like `compare-and-swap` incur high CPU overhead, which potentially leads to low IO utilization of FNDs in out-of-core processing. A well-known alternative that does not require synchronization for each update is message passing technique. FlashGraph [27] adopts message passing by assigning a message queue to each vertex, and assigning each vertex to one of the computation threads based on the vertex ID. FlashGraph pro-



(a) FlashGraph on NAND SSD



(b) FlashGraph on Optane SSD

Fig. 2: **Idle IO periods in FlashGraph on Optane SSD.** Red line: the maximum read bandwidth of Optane SSD. Input graph: rmat30.

cesses these messages at the end of each iteration to update the algorithm-specific vertex data and generate a set of vertices which will be activated in the next iteration.

The problem with the message passing scheme in FlashGraph is the potential risk of *skewed computation* on power-law graphs – some threads need to process more messages than others because certain vertices have much higher number of neighbors than other vertices on these graphs. In out-of-core graph processing, all activities including IO must wait until the straggler thread finishes the processing of messages in each iteration. Crucially, FND can potentially finish all IO requests faster than the straggler thread so it may frequently remain idle over iterations.

We observe this phenomenon on Optane SSD as shown in Figure 2. On NAND SSD (Figure 2 (a)), FlashGraph fully utilizes the device's read bandwidth on three queries, PR, WCC, and SpMV. However, for the same queries, it fails to issue any IO to the Optane SSD at the end of each iteration (the period where the read bandwidth remains zero) due to the straggler thread still processing a large volume of messages (Figure 2 (b)).

Mitigating this problem requires balancing the workload – the messages passed among vertices – between threads but achieving this without synchronization is not straightforward. We solve this problem in Blaze with a synchronization-free, online binning technique we describe in Section IV-A.

B. *Skewed IO*

Another problem that leads to low IO utilization is skewed IO. We observe this problem in Graphene.

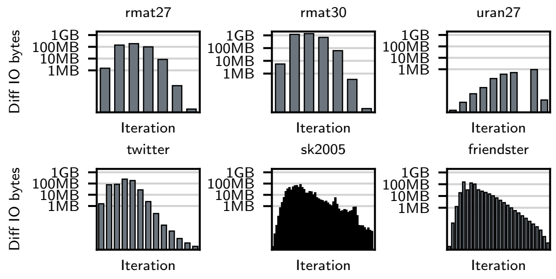


Fig. 3: **Skewed IO in Graphene.** y-axis: $max - min$ IO bytes between eight SSDs for each iteration.

Skewed load of IO across multiple disks is another source of low IO utilization. In synchronous graph processing where edges are distributed in multiple disks, the maximum aggregate IO bandwidth is achieved when all disks are kept busy at all times. When IO is not balanced, however, it leaves some disks to wait until other disks complete their requests. We find that Graphene suffers this *skewed IO* problem due to its topology-aware partitioning scheme.

Graphene adopts 2-D partitioning of a graph with the goal of producing partitions with the same number of edges. Then it distributes these partitions on multiple disks in a way that each disk has the same number of partitions, making each disk have an equal number of edges.

Despite having a balanced partition distribution, Graphene suffers highly skewed IO on algorithms that employ *selective scheduling* of edges. Selective scheduling means that only a subset of the total edges are traversed in a given iteration, a common technique to increase algorithm efficiency by only accessing the necessary edges for a given algorithm goal. In Graphene, these algorithms end up accessing edges on certain disks more than those on others, leading to skewed IO.

Figure 3 shows the skewed IO of Graphene on BFS that employs selective scheduling over iterations. On the y-axis, the figures show the maximum difference between 8 disks in terms of the IO bytes each disk must process in a given iteration. For example, a bar with 10 MB of height means that the disk with the largest amount of IO tasks has to do 10 MB of more IO than the disk with the smallest amount of IO, so the higher bar means IO is more skewed. We observe that Graphene suffers the skewed IO on all power-law graphs. On the uran27 graph with uniform degree distribution, the difference in IO is less than 1 MB. However, on other graphs that follow power-law, the maximum IO difference goes up to more than 100 MB. The impact is more dramatic when considering the ratio, not just the absolute bytes – A disk has to

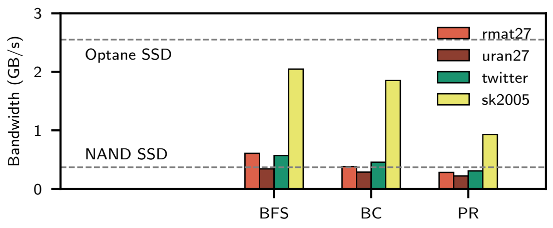


Fig. 4: **Single-threaded graph computation speed (bars) vs. IO bandwidth (lines).**

conduct up to 1.7–2.1× (depending on the input graph except uran27) more IO than another disk.

Based on these results, we conclude that the topology-aware graph partitioning adopted by Graphene incurs the skewed IO problem when running algorithms with selective scheduling. We mitigate this problem with topology-agnostic graph partitioning based on the page interleaving as we describe in Section IV-E.

C. Fast IO, Slow Computation

Graphene's low IO utilization also stems from its thread assignment policy which leads to the *fast producer and slow consumer* problem. Graphene equally devides cores across IO and computation – a pair of cores are assigned to each SSD, one for IO and one for computation. For slow SSDs, this scheme still helps maximize the IO bandwidth by assigning a dedicated thread for each SSD.

However, two threads strictly assigned for each SSD are not sufficient for FNDs because they cannot saturate the bandwidth of FNDs. When the producer (IO thread) sends IO buffers filled with on-disk pages faster than the consumer (computation thread) can process, the free IO buffers soon become unavailable, which in turn blocks the IO thread from issuing more IO requests.

We measure the impact of this problem by comparing the speed of various single-threaded graph computations with the read bandwidth of various storage devices (Figure 4). Compared to slow storage like NAND SSD, single-threaded graph computation is fast enough on a set of workloads. However, it does not keep up with the speed of Optane SSD on all workloads we measured. The result implies that enough threads must be assigned for computation to constantly saturate the underlying FND in out-of-core graph processing.

IV. Blaze Framework

Blaze supports high-performance graph analytics on FNDs by constantly saturating the underlying IO bandwidth, a challenge that was not achieved by current systems. The key to achieving this goal is the low overhead, scatter-gather scheme called *online binning* that processes user-provided graph computations without synchronization while achieving load balance among threads. In addition, Blaze achieves balanced IO among multiple SSDs by partitioning the input graph with page interleaving (RAID 0) that balances IO well on a variety of workloads. Blaze abstracts these mechanisms in the well-known, in-core graph processing API, EDGEMAP and VERTEXMAP [22], to enable the programming of efficient out-of-core graph algorithms without the need to handle complex IO executions.

A. Online Binning

An IO-efficient execution of EDGEMAP relies on low-overhead graph computation which we enable with the technique we call *online binning*. The idea of using bin data structure in graph processing is inspired by propagation blocking [4] but we adapt this idea to out-of-core graph processing.

A bin is a struct kept in DRAM that holds multiple bin records where each bin record is a $\langle vertex\_id, value \rangle$ pair. During execution, Blaze creates a bin record for each algorithm-specific scatter function with the destination vertex id and the value returned by the scatter function. Then Blaze appends the record to the corresponding bin ($bin\_id = vertex\_id \bmod bin\_count$). Once a bin becomes full, Blaze pushes it to a concurrent queue called `full_bins` to allow gather threads to process the records in the full bins. Each gather thread processes one full bin in its entirety. Most importantly, Blaze ensures that *no two gather threads process the same bin at the same time* and this avoids the need to synchronize between gather threads.

To maximize the performance of online binning, Blaze adopts several optimization techniques. First, Blaze uses a small fixed size, per-CPU buffer [4] to reduce the synchronization overhead while binning, where the buffer allocates memory space for each bin. Blaze first appends the bin record to this buffer, and once it becomes full, Blaze copies the records in the buffer to the corresponding bin in batch. Second, Blaze uses MPMC queue for `full_bins` for highly concurrent push/pull of full bins between scatter and gather threads. Third, Blaze implements each bin as a pair of bins to ensure the forward progress of both scatter and gather threads. Once one of the pair becomes full, its pointer is appended to the `full_bins` queue for gather threads while the other bin serves scatter threads. A scatter thread is blocked until a gather thread finishes the processing of the full bin and returns it to the empty state.

Online binning has several configuration parameters including bin count, bin size, and the ratio

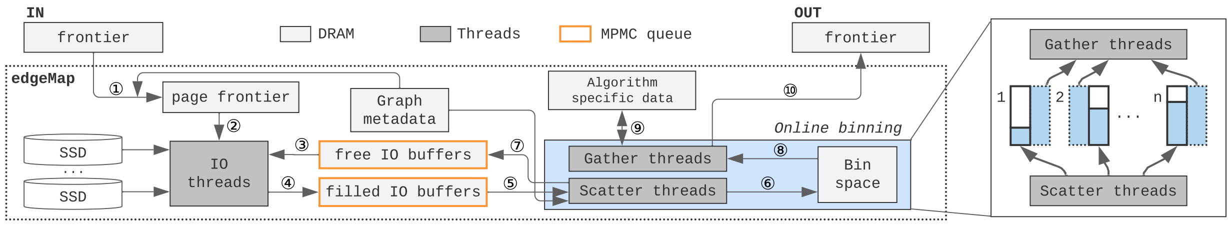


Fig. 5: **Out-of-core EDGEMAP engine in Blaze.**

between scatter and gather threads. Data in Section V-E shows that performance is robust across a wide range of values, so precise tuning is not required. In particular, our data show that *one thousand bins, 0.05× of the input graph size for bin space*, and *an equal number of scatter and gather threads* will provide good performance in general and that more careful tuning improves performance by, at most, 5%.

B. Programming API

To support the programming of efficient out-of-core graph algorithms, Blaze provides two key APIs, EDGEMAP and VERTEXMAP. The APIs were first introduced by Ligra [22] in-core graph processing framework and have shown that they can be used to express a broad range of efficient, parallel graph algorithms [22], [21], [8] for in-memory graph processing. We extend them to enable efficient out-of-core graph processing while hiding the binning-based execution entirely from the user.

$$\begin{aligned}\textbf{EDGEMAP}(&graph : Graph,\\ &frontier : VertexSubset,\\ &f_s : (vertex \times vertex) \to value\_type,\\ &f_g : (vertex \times value\_type) \to bool,\\ &cond : vertex \to bool,\\ &output : bool) : VertexSubset\end{aligned}$$

Executes two edge functions, f_s and f_g, to the edges whose source vertices are in the given $frontier$. Users provide the scatter function f_s that returns an algorithm-specific value to scatter it to neighboring vertices. The value is scattered to the gather threads only when *cond* returns true with the destination vertex ID as argument. Users also provide the gather function f_g that accumulates the scattered values to the algorithm-specific data array. When the *output* is true, EDGEMAP creates an output frontier and pushes the destination vertex ID to the frontier if f_g returns true.

The scatter and gather functions communicate intermediate data via bin data structure provided by the online binning mechanism. This ensures that scatter and gather steps are executed without synchronization overhead while achieving load balance among worker threads.

$$\begin{aligned}\textbf{VERTEXMAP}(&frontier : VertexSubset,\\ &f : vertex \to bool) : VertexSubset\end{aligned}$$

Applies a vertex function f to each vertex in the $frontier$. It conditionally filters out the vertices from the frontier when f returns true, and returns a new frontier. In Blaze, VERTEXMAP executes entirely in memory as all vertex-related data is placed in memory. In most algorithms, VERTEXMAP is used along with EDGEMAP alternatively in an iteration to update vertex values and reduce the next frontier size. In Section IV-D, we describe how BFS, PageRank, and WCC algorithms use both EDGEMAP and VERTEXMAP functions together in more detail.

C. *Out-of-core* EDGEMAP *Execution*

Figure 5 shows the architecture of Blaze's EDGEMAP engine and how an EDGEMAP function is executed in an out-of-core fashion along with online binning.

With the frontier as input, an EDGEMAP function starts execution by first transforming the given frontier into the $pagefrontier$, a data structure that contains the disk page IDs that contains the target vertex IDs in the frontier (step 1). Blaze uses all available threads to accelerate this transformation before starting issuing IO requests. Once the page frontier is ready, IO threads start fetching the page IDs from it and send IO requests to the underlying SSDs (step 2) with the free IO buffers (step 3). Blaze uses one thread for each SSD and maintains the page frontier for each SSD. Once the corresponding disk pages are fetched into the buffers (step 4), online binning comes into play – the scatter threads get these fill buffers (step 5), append the records to the corresponding bins (step 6), and return the IO buffers back to the free IO buffer pool (step 7). Concurrently, the gather threads fetch the *full* bins (step 8) and apply the records in the bins into the algorithm-specific, vertex data (step 9). Finally, the gather threads returns a new frontier if required by the caller of EDGEMAP.

To support the fast communication of IO buffers between IO threads and computation threads (scatter, gather threads), Blaze uses a concurrent MPMC (multi-producer, multi-consumer) queue. Blaze maintains two queues, one for free IO buffers, and the other for filled IO buffers, each of which contains the address of the buffer page.

Blaze uses two types of frontier, $VertexSubset$ and $PageSubset$ for the vertex frontier and page frontier, respectively. Both types abstract the sparse and dense format and switch between them internally depending on the density of the members. Both types use concurrent set data structure when the members are sparse and use bitmap when the members are dense, similarly in Ligra [22]. $PageSubset$ is only used internally for IO and not exposed to the users.

For IO execution, Blaze issues IO requests based on the page IDs contained in the page frontier. For continuous pages, Blaze issues only *small contiguous IO* unlike FlashGraph [27] and Graphene [16]. Concretely, Blaze merges up to four contiguous 4 kB pages as larger IO request is not beneficial on FND. Rather, it is studied in Graphene [16] that the large IO significantly increases the Asynchronous IO submission time. Also, Blaze does not attempt to merge non-consecutive pages even if they are within a certain threshold [16] – On FNDs, 4 kB random IO is already fast enough such that there is little incentive to issue large IO requests at the cost of accessing non-target pages.

D. Examples

BFS Algorithm 1 shows a parallel out-of-core BFS algorithm written in Blaze's API. The user provides two edge functions, SCATTER and GATHER. Leveraging the online binning internally, they cooperatively update the *Parent* array without synchronization overhead. Specifically, SCATTER function examines input edges and returns the source vertex ID. To reduce unnecessary propagation of values, the user also provides COND function – SCATTER is executed only when the destination vertex has not been visited yet by checking if COND returns true. Then GATHER function receives the value (v) along with the associated destination ID (d). If the destination vertex has not been visited ($Parent[d] == -1$), GATHER updates the parent array with the source vertex ID and returns 1, activating the current destination vertex in the next iteration. Finally, these functions are used in the main BFS function as arguments to the EDGEMAP that iteratively runs until the frontier F becomes empty.

Algorithm 1 Breadth-First Search

```
1: $Parent = \{-1, ..., -1\}$                    ▷ initialize all to -1's
2:
3: procedure SCATTER($s$, $d$)                   ▷ scatter function
4:     return $s$
5: end procedure
6:
7: procedure GATHER($d$, $v$)                    ▷ gather function
8:     if $Parent[d] == -1$ then
9:         $Parent[d] = v$
10:        return 1
11:    end if
12:    return 0
13: end procedure
14:
15: procedure COND($d$)                          ▷ conditional function
16:    return $Parent[d] == -1$
17: end procedure
18:
19: procedure BFS(G, s)                          ▷ s is the root
20:    $Parent[s] = s$
21:    $F = \{s\}$
22:    while $\neg F.empty()$ do
23:        $F = $ EDGEMAP($G, F$, SCATTER, GATHER, COND, $true$)
24:    end while
25: end procedure
```

PageRank Algorithm 2 shows an example of PageRank that implements PageRank-delta algorithm [22], [17], a variant of PageRank in which vertices are active in an iteration only if they have accumulated enough change in their page rank values. In our implementation, EDGEMAP propagates the delta value of each vertex, normalized with its out-degree, to the out-going neighbors in SCATTER and accumulates those values in GATHER without synchronization. Then VERTEXMAP applies the accumulated delta values kept in ngh_sum to the $delta$ array and filters out vertices whose change in the page rank value in p is less than a given threshold e, as implemented in APPLYFILTER. The EDGEMAP and VERTEXMAP alternately run until no vertex is active in the frontier.

WCC Algorithm 3 shows the shortcutting label propagation algorithm running on an undirected graph [23] implemented in Blaze API. While SCATTER and GATHER updates the destination vertex value with the smaller vertex ID (normally as in original label propagation), the shortcutting mechanism in APPLYFILTER conducts pointer jumping to accelerate the label propagation. In addition, it activates only the vertices that suffered the value change from the previous iteration. With Blaze API, our WCC implementation executes EDGEMAP for both CSR ($outG$) and a transpose of it (inG) to propagate vertex values on an undirected graph. The algorithm finishes when no further propagation is

Algorithm 2 PageRank

```
G ← $Input\ graph$
$p = \{0, ..., 0\}$
$ngh\_sum = \{0, ..., 0\}$
$delta = \{\frac{1}{V}, ..., \frac{1}{V}\}$
$D \leftarrow 0.85, e \leftarrow threshold$

procedure SCATTER($s$, $d$)                          ▷ scatter function
    return $delta[s]/G.get\_degree(s)$
end procedure

procedure GATHER($d$, $v$)                           ▷ gather function
    $ngh\_sum[d] += v$
    return 1
end procedure

procedure COND($d$)                                  ▷ conditional function
    return 1
end procedure

procedure APPLYFILTER($i$)                           ▷ vertex function
    $delta[i] = ngh\_sum[i] * D$
    $ngh\_sum[i] = 0$
    if $|delta[i]| > e * p[i]$ then
        $p[i] += delta[i]$
        return 1
    else
        return 0
    end if
end procedure

procedure PAGERANK(G)
    $F = \{1, ..., 1\}$                              ▷ activate all vertices
    while $\neg F.empty()$ do
        EDGEMAP($G$, $F$, SCATTER, GATHER, COND, $false$)
        $F$ = VERTEXMAP($F$, APPLYFILTER)
    end while
end procedure
```

Algorithm 3 WCC

```
$Ids = \{0, ..., V-1\}$                             ▷ initialize all to node ids
$PrevIds = \{0, ..., V-1\}$                         ▷ initialize all to node ids

procedure SCATTER($s$, $d$)                          ▷ scatter function
    return $Ids[s]$
end procedure

procedure GATHER($d$, $v$)                           ▷ gather function
    $orig\_id = Ids[d]$
    if $v < orig\_id$ then
        $Ids[d] = v$
    end if
    return 1
end procedure

procedure COND($d$)                                  ▷ conditional function
    return 1
end procedure

procedure APPLYFILTER($i$)                           ▷ vertex function
    $id = Ids[Ids[i]]$
    if $Ids[i]! = id$ then
        $Ids[i] = id$
    end if
    if $PrevIds[i]! = Ids[i]$ then
        $PrevIds[i] = Ids[i]$
        return 1
    else
        return 0
    end if
end procedure

procedure WCC(outG, inG)
    $F = \{1, ..., 1\}$                              ▷ activate all vertices
    while $\neg F.empty()$ do
        EDGEMAP($outG$, $F$, SCATTER, GATHER, COND, $false$)
        EDGEMAP($inG$, $F$, SCATTER, GATHER, COND, $false$)
        $F$ = VERTEXMAP($F$, APPLYFILTER)
    end while
end procedure
```

required.

E. Balanced IO

In addition to the low-overhead, balanced computation powered by online binning, Blaze also achieves balanced IO with *page-interleaved, Compressed Sparse Row (CSR) format* – Blaze stripes a CSR graph into multiple SSDs in 4 kB granularity. Page interleaving (RAID 0) is a well-known technique used in various HPC domains to maximize the aggregate bandwidth of underlying devices. We find that it is also effective in out-of-core graph analytics. We reject other topology-aware partitioning schemes such as 2-D partitioning used in Graphene [16] as they incur imbalanced load across multiple disks when only a subset of edges need to be accessed in each iteration.

F. Memory Usage

Blaze requires the memory space as follows except the algorithm-specific data.

System-level Regardless of any given workload, Blaze requires a static memory space to allocate IO buffers from. In Blaze, large memory space is not required for IO buffers as scatter threads return the IO buffers quickly enough for IO threads to reuse those buffers. Accordingly, we set the memory space relatively small (64 MB in all workloads) compared to the input graph size we tested.

In addition, Blaze requires a memory space to maintain bins for online binning. We experimentally decide the proper bin size based on our study in Section V-E.

Graph metadata For a given input graph, Blaze maintains an index array and a key-value map in memory for efficient graph access. To keep the graph index array compact, Blaze uses indirection as in Figure 6 – Blaze groups sixteen 4 bytes-sized degrees into a single cache-line in the *degrees* region and only keeps the location of the cache-lines in the *offsets* region. With indirection, the offset is retrieved by first looking up the offsets region with $vertex\_id$ / 16 as key then adding degrees up to $vertex\_id$ % 16 in the corresponding cache-line in degrees region. This indirection-based index array in Blaze requires about $4\ bytes \times |V|$ of memory.

In addition to the index, Blaze keeps an addi-

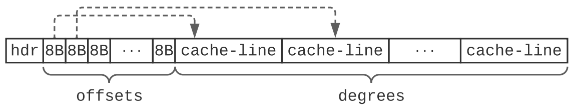


Fig. 6: **Indirection-based graph index in Blaze.**

tional map *page-to-vertex map* to accelerate access to vertex data given a page number. The map returns a $\langle begin\_vertex\_id, end\_vertex\_id \rangle$ pair given an on-disk page number as key. The size of this structure is small as it only requires 8 bytes for each disk page.

V. Evaluation

We evaluate Blaze with a variety of workloads, comparing it against two state-of-the-art, open-sourced out-of-core graph processing systems, FlashGraph [27] and Graphene [16]. In addition, we study how Blaze scales with more hardware resources, and how it performs with different binning configurations.

A. Experimental Setting

Target queries We use the following graph algorithms to evaluate Blaze.

- Breadth-First Search (BFS)
- PageRank (PR) using the delta variant algorithm [17].
- Weakly Connected Components (WCC) using Label propagation [28].
- Sparse Matrix-Vector Multiplication (SpMV)
- Betweenness Centrality (BC) using Brandes's algorithm [7].

We implement these queries based on the implementations in Ligra [22] as both systems share the same API. The difference is that Blaze algorithms require the scatter and gather functions as input to the EDGEMAP function while Ligra requires providing only a single, synchronization-based edge function to the EDGEMAP. Also, Ligra's EDGEMAP is executed purely in memory while Blaze newly introduces the execution of EDGEMAP in an out-of-core fashion.

System configuration Our testbed is a single socket, Intel Xeon Gold 6230 processor (2.1 GHz) with 20 physical cores (no hyperthreading). The machine is equipped with 96 GB of DRAM; one 1.9 TB Intel NAND SSD (DC S3520) and one 960 GB Intel Optane SSD (DC P4800X).

B. Comparison with Other Systems

We compare the performance of Blaze with FlashGraph and Graphene on six input graphs stored on an Intel Optane SSD. Among the five target queries – BFS, PR, WCC, SpMV, and BC, we could

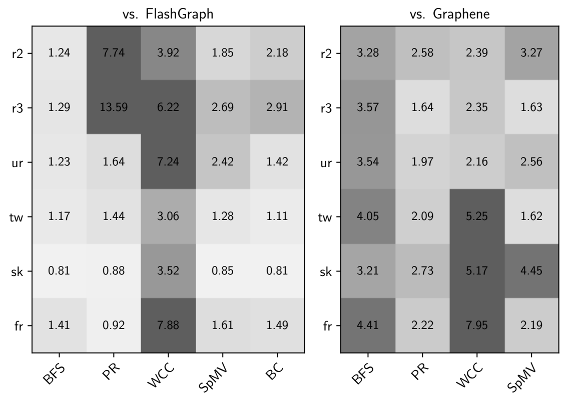
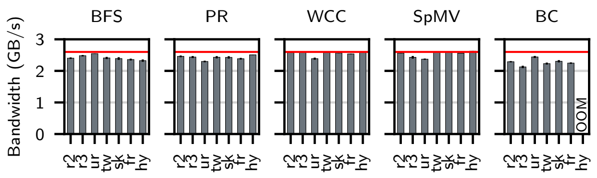
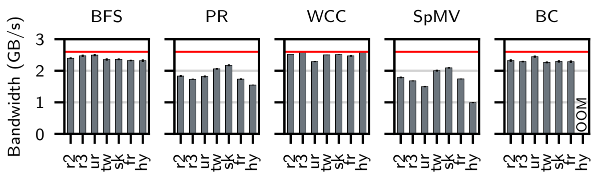


Fig. 7: **Speedups over FlashGraph and Graphene.**



(a) Blaze



(b) Synchronization-based variant

Fig. 8: **Average read bandwidth on Optane SSD**

not compare the result of BC with Graphene since Graphene does not implement BC. For all experiments, we used 16 threads from a single socket for fair comparison.

Figure 7 shows the speedup of Blaze over FlashGraph (left) and Graphene (right). Except on sk2005, Blaze generally outperforms FlashGraph, achieving up to 13.6× speedup when running PageRank on rmat30 graph. On sk2005, Blaze performs 12–20% slower than FlashGraph because the sk2005 graph has a high locality [5] such that storage access is minimized by hitting the page cache implemented in FlashGraph with LRU policy. Blaze only implements the random eviction of IO buffer pages, and we leave implementing more advanced eviction policies as future work.

On the other hand, Blaze consistently outperforms Graphene with 1.6–7.9× of speedups on our target workloads. In case of PR, we compare the execution time of 1 PR iteration as Graphene does not implement PR with selective scheduling.

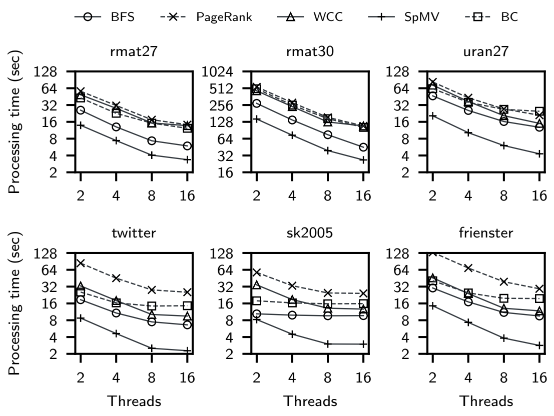


Fig. 9: **Thread scaling**

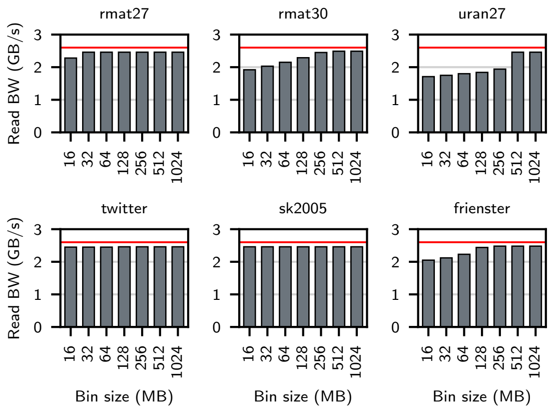


Fig. 10: **Impact of binning space.**

C. IO Utilization

Blaze achieves high IO bandwidth close to the FND's device bandwidth. In Figure 8, we report the average IO bandwidth of our target workloads measured on Optane SSD. We calculate the average bandwidth as the total read IO bytes divided by the total query execution time. To see the impact of online binning on IO utilization, we compare Blaze with a synchronization-based variant of Blaze that uses atomic operations like `compare-and-swap` to synchronize parallel updates.

Unlike FlashGraph and Graphene which underutilize an Optane SSD's bandwidth as reported in Figure 1, Blaze almost fully utilizes the bandwidth on all our workloads. On computation-heavy workloads like PageRank and SpMV, the high IO bandwidth is only achieved with online binning. Otherwise, the synchronization-based Blaze achieves only 38–85% of the Optane's device bandwidth on both queries depending on the workload.

D. Scalability

Blaze scales with increasing core count as long as the underlying storage is not saturated. Figure 9 (with both axes in log scale) shows how Blaze's performance scales when running our workloads on a single Optane SSD. Performance almost linearly scales with more cores on most of the workloads. On a certain set of workloads (e.g., BFS on sk2005), using one scatter and one gather thread (therefore two cores) is sufficient to saturate the IO bandwidth as the graph has high locality and thus causes less CPU overhead with processor cache hits. In these cases, Blaze does not scale with more threads as the IO bandwidth becomes the bottleneck.

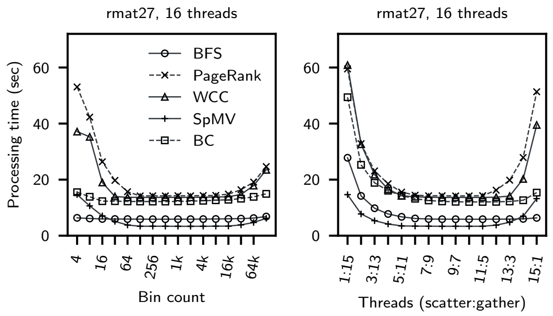


Fig. 11: **Impact of binning configurations.**

E. Impact of Online Binning Configurations

Online binning in Blaze requires users to set up a few parameters to perform as expected.

Bin size The total bin size must be set large enough not to slow down the performance of Blaze. To understand the right bin size, we measured the average read bandwidth of SpMV query on all input graphs while varying the total bin size from 16 MB to 1 GB. Based on the result in Figure 10, we find that a good heuristic value for the bin size is roughly $\frac{1}{20}|E| \times 4\ bytes$ for each graph but using smaller values is also viable on some graphs.

Bin count We also study how different bin counts can impact the performance of online binning. For this, we measure the processing time of all queries on rmat27 graph with 256 MB of bin space while doubling the bin count from 4 to 131072. Figure 11 shows that the processing time is relatively stable for a large range of bin counts but increases significantly when the value is too large or too small.

Scatter, gather thread ratio An intuition in choosing the right ratio between the number of scatter and gather threads is to consider the relative computation load between scatter and gather in each algorithm. If the load is similar, a good choice is to use an equal number of threads for both tasks.

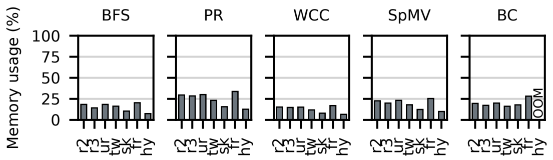


Fig. 12: **Memory footprint relative to the input graph size.**

The result in Figure 11 reflects this intuition – The execution time remains constantly low when a similar amount of threads are used for scatter and gather but sharply increases as more threads are assigned for one task than the other.

F. Memory Usage

Under the semi-external model, Blaze aims to minimize the memory consumption while supporting high-performance out-of-core graph processing on FNDs. Figure 12 shows the memory consumption of Blaze on our target workloads. The figure depicts the ratio of memory footprint to input graph size in each workload.

Depending on the workload, Blaze's memory footprint is 10–34% of the input graph size. Concretely, memory footprint is affected by the input query and the underlying graph. For BFS that requires only a single integer array to keep the parent relationships (Algorithm 1), the ratio is 10–20% throughout different input graphs.

However, certain queries require more memory due to the nature of their algorithms. For instance, in case of our PageRank implementation, the ratio goes up to 16–33% as PageRank-delta requires three floating point arrays to implement (Algorithm 2). In addition, we were not able to run BC on hyperlink2014 graph because Brandes's algorithm [7] requires more than 96 GB of memory to run on 512 GB of undirected hyperlink2014 graph. We expect mitigating these problems require more memory-efficient algorithms.

Except for BC, Blaze can successfully run other queries on hyperlink2014 with the limited amount of memory compared to the graph size while existing in-core frameworks [22], [19], [26] will run into the out-of-memory issue with this dataset.

VI. Related Work

The design of graph processing systems has been evolved mainly in three different ways depending on the use of secondary storage and the use of memories in multiple machines.

In-core graph processing processes graphs entirely in the main memory of a single machine. Galois [19] is a lightweight runtime that offers API for implementing efficient task scheduling of various graph algorithms. GAP [6] offers a benchmark suite of various in-core graph algorithms to standardize the in-core graph processing evaluations. GraphIt [26] introduces a new domain-specific language and runtime to help separate the process of algorithm writing and performance optimization. Finally, Ligra [22] is an in-core graph processing framework that offers simple APIs for writing graph algorithms and optimizes the execution of those algorithms by automatically switching between push and pull-based operations based on a user-provided threshold. Blaze extends the Ligra's APIs but adapts them to support efficient out-of-core execution.

Out-of-core graph processing uses storage devices to hold the large graphs that do not fit in a single machine's memory and process those graphs by judiciously moving pages between the storage and the memory. To minimize the overhead of expensive IO, the design of out-of-core systems has evolved in step with the performance characteristics of underlying storage devices. GraphChi [15] was the first out-of-core graph processing system designed for sequential disks. XStream [20] explored a new tradeoff that fully exploits the benefit of sequential access at the cost of more IO. More out-of-core systems have been further developed to leverage the sequentiality of disks [29], [12], [24].

On the other hand, systems like FlashGraph [27] and Graphene [16] explored the ways to utilize fast random IO that early-generation SSDs offer, achieving significant performance improvement over prior systems as less IO is required. Blaze makes a similar design choice but further optimizes software mechanisms to constantly saturate the high IO bandwidth of modern FNDs. A more recent work by Elyasi *et al.* [9] explored a new graph partitioning technique to leverage the fast random IO of FNDs in the fully-external processing style. However, this work makes a different tradeoff from Blaze – it places only a subset of vertex data in memory to achieve smaller memory footprint but potentially at the cost of performance. Unfortunately, we could not compare this work with Blaze as it is not public.

Distributed graph processing is another way to process large graphs by holding them in the memories of multiple machines and processing them via network communications. Systems like PowerGraph [10], Pegasus [13], and GraphLab [17] have been developed to ease the programming of distributed graph algorithms while offering high performance on highly-skewed graphs. Nonetheless, distributed graph processing systems often lack efficiency with low per-CPU performance [18] and do not necessarily outperform out-of-core systems despite using more resources from a cluster of machines [15], [29], [27], [16].

For higher scalability in terms of both CPU and storage, Blaze could be further scaled out on multiple machines where each machine is equipped with one or more FNDs. One potential way to scale out Blaze is to partition the input graph based on the *destination vertex* and place each partition in each machine. This allows a single machine to process only a subset of edges and vertex-related values, and, more importantly, to propagate values between scatter and gather threads *locally*, avoiding the costly network communications during EDGEMAP execution. We leave scaling out Blaze in this manner as future work.

VII. Conclusion

We present Blaze, a new out-of-core graph processing system optimized for FNDs. Blaze offers high-performance graph analytics by constantly saturating FNDs with a novel scatter-gather technique called online binning while previous techniques like synchronization or message passing fail to achieve this goal. Blaze offers succinct APIs for writing efficient, out-of-core graph algorithms without the burden to deal with complex IO executions.

Acknowledgments

This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We thank the anonymous reviewers for their valuable feedback.

References

[1] Intel optane ssd 9 series. https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html.

[2] Samsung v-nand ssd. https://www.samsung.com/us/business/computing/memory-storage/solid-state-drives/explore/.

[3] Samsung z-ssd. https://semiconductor.samsung.com/ssd/z-ssd/.

[4] S. Beamer, K. Asanović, and D. Patterson. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 820–831, 2017.

[5] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, IISWC '15, page 56–65, USA, 2015. IEEE Computer Society.

[6] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[7] Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[8] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. Sage: Parallel semi-asymmetric graph algorithms for nvrams. *Proc. VLDB Endow.*, 13(9):1598–1613, May 2020.

[9] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-scale graph processing on emerging storage devices. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 309–316, USA, 2019. USENIX Association.

[10] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 17–30, USA, 2012. USENIX Association.

[11] 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, page 77–85, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 411–424. IEEE Press, 2018.

[13] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. *2009 Ninth IEEE International Conference on Data Mining*, pages 229–238, 2009.

[14] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast nvm storage with udepot. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 1–15, USA, 2019. USENIX Association.

[15] 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, page 31–46, USA, 2012. USENIX Association.

[16] Hang Liu and H. Howie Huang. Graphene: Fine-grained IO management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, Santa Clara, CA, February 2017. USENIX Association.

[17] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[18] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, page 14, USA, 2015. USENIX Association.

[19] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456–471, New York, NY, USA, 2013. Association for Computing Machinery.

[20] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery.

[21] Julian Shun. *An Evaluation of Parallel Eccentricity Estimation Algorithms on Undirected Real-World Graphs*, page 1095–1104. Association for Computing Machinery, New York, NY, USA, 2015.

[22] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Pro-*

ceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’13, page 135–146, New York, NY, USA, 2013. Association for Computing Machinery.

[23] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. Shortcutting label propagation for distributed connected components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, WSDM ’18, page 540–546, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Keval Vora. LUMOS: Dependency-driven disk-based graph processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 429–442, Renton, WA, July 2019. USENIX Association.

[25] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for inmemory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI’12, page 2, USA, 2012. USENIX Association.

[26] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[27] 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)*, pages 45–58, Santa Clara, CA, February 2015. USENIX Association.

[28] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, 2002.

[29] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

1 SYSTEM REQUIREMENT

1.1 Hardware dependencies

We evaluated Blaze with the following machine configuration.

- CPU : Intel Xeon Gold 6230 processor (2.1 GHz) with 20 physical cores in a single socket (no hyperthreading)
- Memory : 96GB of DRAM
- Storage : 1.9TB Intel NAND SSD (model: DC S3520), 960GB Intel Optane SSD (model: DC P4800X)
- OS: Linux 5.12

Blaze is not yet optimized for multi-socket processors so we recommend to use single-socket machine or similar setting to reproduce similar results in the paper.

Also, while Blaze runs on any type of block device, its high performance is best judged when running on fast NVMe SSDs such as Intel Optane SSD. If desired, we will provide proper guidelines to allow access to our testbed.

1.2 Software dependencies

We provide a docker image containing pre-built binaries so it is not necessary to explicitly build the dependencies to run Blaze. As for the docker version, we used the version 20.10.10, so we expect the same or a newer version to work.

1.3 Data sets

We evaluated Blaze's performance on six input graphs – rmat27, rmat30, uran27, twitter, sk2005, and friendster.

To download each dataset,

```
$ wget https://storage.googleapis.com/nvsl-aepdata/graphdata/sc22/{dataset_name}.zip
```

The size of each zip file is as follows.

- rmat27: 13GB
- rmat30: 102GB
- uran27: 16GB
- twitter: 8.5GB
- sk2005: 2.3GB
- friendster: 13GB

After unzipping, each dataset consists of four files: Two of them are *.gr.index* (index file) and *.gr.adj.0* (adjacency list file), collectively representing a CSR format that stores outgoing neighbors of each vertex. Additional two files, *.tgr.index* and *.tgr.adj.0*, represent a transpose of the given CSR graph.

2 INSTALLATION

2.1 Storage setup

The target storage should be mounted in the target machine to place the input graphs. An example of mounting our target disk /dev/nvme0n1 to /mnt/nvme using Ext4 file system is as follows.

```
$ sudo mkfs.ext4 -F /dev/nvme0n1
$ mkdir -p /mnt/nvme
$ sudo mount /dev/nvme0n1 /mnt/nvme
```

Then, place the downloaded input graphs under /mnt/nvme.

2.2 Getting inside the docker container

As Blaze binaries are available in the provided docker image, it is necessary to run and get into the container as follows.

```
$ docker run –rm -it -v "/path/to/your/storage":"/mnt/nvme" junokim8/blaze:1.0 /bin/bash
```

Inside the docker container console, the Blaze binaries are available at /home/blaze/build.

3 EVALUATION WORKFLOW

3.1 Major claims

The major claims and key results made in Blaze paper are listed as follows.

(1) Blaze outperforms FlashGraph and Graphene with significant speedups [Figure 7].
(2) Blaze's online binning mechanism is the key to saturating Intel Optane SSD [Figure 8].
(3) Blaze scales well with more threads as long as IO is not saturated [Figure 9].

3.2 Reproducing results

To reproduce the above figures, we provide an automated benchmark script that generates CSV file for each figure. They are available under "/home/blaze/scripts" within the container.

To reproduce results of each figure, do the following.

```
$ ./run_figure7.sh (30 minutes)
$ ./run_figure8.sh (1 hour)
$ ./run_figure9.sh (3 hours)
```

This will generate csv file(s) corresponding to each figure.

3.3 Running each workload explicitly

For instance, run the following command to run BFS on rmat27 graph. This example calculates BFS using 17 threads (16 for computation and 1 for IO) starting from vertex 0.

```
$ ./bin/bfs -computeWorkers 16 -startNode 0 /mnt/nvme/rmat27.gr.index /mnt/nvme/rmat27.gr.adj.0
```

Certain queries require a transpose of the input graph as well. For instance, our Betweenness Centrality implementation falls into this case. To give the transpose graph as additional input, use "-inIndexFilename" and "-inAdjFilenames" as follows.

```
$ ./bin/bc -computeWorkers 16 -startNode 0 /mnt/nvme/rmat27.gr.index /mnt/nvme/rmat27.gr.adj.0 -inIndexFilename /mnt/nvme/rmat27.tgr.index -inAdjFilenames /mnt/nvme/rmat27.tgr.adj.0
```

The number of IO thread is automatically determined by Blaze depending on the number of given partitions. In the above examples, we use one partition.

3.4 Configuring binning

Each graph query binary provides options to customize binning configuration such as "-binSpace", "-binningRatio", and "-binCount". In our steps for generating figure7, 8, and 9, we used "-binSpace=256", "-binningRatio=0.5", and "-binCount=1024".

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1
Persistent ID: `junokim8/blaze:1.0`
Artifact name: docker image for evaluating Blaze

Artifact 2
Persistent ID: `10.5281/zenodo.6976001`
Artifact name: DOI

Artifact 3
Persistent ID: `https://github.com/NVSL/blaze`
Artifact name: Github URL

Reproduction of the artifact with container: Download the docker image as follows:

$ docker pull junokim8/blaze:1.0

Then run the container as follows:

$ docker run –rm -it -v "/path/to/your/storage":"/mnt/nvme" junokim8/blaze:1.0 /bin/bash

The instructions for reproducing results of the paper are described in the above section.