Approaching DRAM performance by using microsecond-latency flash memory for small-sized random read accesses: a new access method and its graph applications

Tomoya Suzuki, Kazuhiro Hiwada, Hirotsugu Kajihara, Shintaro Sano, Shuou Nomura, Tatsuo Shiozawa
Kioxia Corporation, Japan
{tomoya.suzuki,kazuhiro.hiwada,hirotsugu.kajihara,shintaro.sano,shuou.nomura,tatsuo.shiozawa}@kioxia.com

ABSTRACT

For applications in which small-sized random accesses frequently occur for datasets that exceed DRAM capacity, placing the datasets on SSD can result in poor application performance. For the read-intensive case we focus on in this paper, low latency flash memory with microsecond read latency is a promising solution. However, when they are used in large numbers to achieve high IOPS (Input/Output operations Per Second), the CPU processing involved in IO requests is an overhead. To tackle the problem, we propose a new access method combining two approaches: 1) optimizing issuance and completion of the IO requests to reduce the CPU overhead. 2) utilizing many contexts with lightweight context switches by stackless coroutines. These reduce the CPU overhead per request to less than 10 ns, enabling read access with DRAM-like overhead, while the access latency longer than DRAM can be hidden by the context switches. We apply the proposed method to graph algorithms such as BFS (Breadth First Search), which involves many small-sized random read accesses. In our evaluation, the large graph data is placed on microsecond-latency flash memories within prototype boards, and it is accessed by the proposed method. As a result, for the synthetic and real-world graphs, the execution times of the graph algorithms are 88-141% of those when all the data are placed in DRAM.

PVLDB Reference Format:
Tomoya Suzuki, Kazuhiro Hiwada, Hirotsugu Kajihara, Shintaro Sano, Shuou Nomura, Tatsuo Shiozawa. Approaching DRAM performance by using microsecond-latency flash memory for small-sized random read accesses: a new access method and its graph applications. PVLDB, 14(8): 1311-1324, 2021.
doi:10.14778/3457390.3457397

1 INTRODUCTION

In data-intensive applications, accessing data often limits the performance of the entire application. Applications with data size exceeding DRAM capacity need to put the data outside of DRAM such as SSD, but access to such data is slower than DRAM. For example, the load latency of DRAM is tens of nanoseconds, while the read latency of a standard SSD is tens of microseconds. It is several hundred times longer than DRAM, which can negatively affect the application performance. If the access to the data is sequential, it is relatively easy to achieve high performance even by using SSD thanks to prefetching and adequate bandwidth. Since the access pattern is known, the prefetching and buffering in large units works well to hide the read latency. In addition, bandwidth of several GB/s to tens of GB/s can be provided by using multiple modern SSDs. In some cases, even if the data is placed on SSDs, it is possible to achieve the application performance close to that of *in-memory* in which all the data are placed on DRAM [35, 48].

Small-sized random accesses to SSD can degrade the application performance compared to in-memory. As the locality of accesses decreases, caching mechanism becomes less effective and random IOPS performance is more likely to have a direct impact on execution time. Existing SSDs have much lower random IOPS performance than DRAM. By using low latency flash memory such as [15], IOPS performance can be improved by an order of magnitude. By deploying a lot of low latency flash memories, random read access performance about 100 MIOPS can be provided at the hardware level. Although this is not comparable to the peak random IOPS performance of DRAM, applications with an IOPS requirement of tens of MIOPS may achieve the execution time close to that of DRAM. However, when flash memory is used, CPU processing to access the memory can degrade application performance in this IOPS range as shown below. Due to the small-sized random access, it is necessary to issue a request every time to read a small piece of data such as several to several tens of bytes, but this request itself incurs an overhead for CPU. Although the CPU overhead can be small by using direct access from user space such as SPDK [9], it still requires about 100 ns of CPU time per request [14]. Furthermore, it is often difficult to hide the SSD's latency by prefetching due to random accesses. Context switch is a conventional technique for hiding the latency. However, context switch by OS takes a few microseconds [34], and tens of nanoseconds even in user space [21]. In order to achieve performance close to that of DRAM, the total time of processing for issuing requests, context switches, and other overheads must be close to the stall time waiting for DRAM response.

Recently, SCM (Storage Class Memory) has been used as a solution for the dataset that exceeds DRAM capacity. In this paper, SCM refers only to byte-addressable memory. The Intel® Optane™ DC Persistent Memory Module (DCPMM) [5] is one of the currently commercialized SCMs. The DCPMM has a random load access latency over 300 ns [25], which is shorter than SSD but longer

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 8 ISSN 2150-8097.
doi:10.14778/3457390.3457397

than DRAM. Due to out-of-order execution by CPU, a load instruction to the DCPMM and other instructions can be executed in an overlapping manner, but it is often difficult to completely hide the latency. Therefore, simply placing data in DCPMM instead of DRAM and using load instructions to access it may degrade application performance [42]. Software level efforts may be required to optimize for near DRAM performance using SCM.

In this paper, we present a new solution for small-sized random read access using flash memory rather than byte-addressable SCM. Although the access latency of flash memory is longer than that of typical SCMs and microsecond-level even when low latency flash memory is used, we show that performance close to in-memory can be achieved. In this paper we focus on read-intensive workloads, which includes important applications such as graph analytics or KVS (Key Value Store). We have developed an efficient access method for the read accesses that is different from existing methods of accessing SSDs. We present the following two points about the new access method. (1) We propose a new hardware interface that reduces the CPU overhead of read request issuance and completion compared to existing SSDs (NVMe™ SSDs). The CPU processing time per request is in less than 10 ns, which may be shorter than CPU stall cycles for DRAM access. (2) Hundreds of contexts are used per core to hide microsecond latency. In order to make the CPU overhead of switching between contexts negligible, we utilize stackless coroutines. Using multiple prototype boards, XL-FLASH™ Demo Drive (*XLFDD*) [46], equipped with low latency flash memories and implementing the new interface, we show in actual system that 99 MIOPS can be achieved with single core while hiding microsecond read latency.

We apply the proposed method to BFS-like graph algorithms such as BFS, BC (Betweenness Centrality) and SSSP (Single Source Shortest Path). These algorithms share a commonality that they start from a given set of vertices and then recursively traverse their neighboring vertices, whose characteristics are called *BFS-like* in [32]. They tend to have many small-sized random accesses and poor locality. The required random IOPS for the algorithms is in the range of a several MIOPS to 200 MIOPS (estimated from the results of in-memory executions), depending on the graph algorithm and the input graph. The existing out-of-DRAM solutions cannot handle them efficiently [35]. In this paper, we show that when using XLFDD, these graph algorithms can be processed in the execution time close to that of using DRAM. To compare the performance with in-memory and XLFDD, we use the existing in-memory graph processing implementation, GAP benchmark [18]. We put the large portion of the graph data on XLFDD instead of DRAM, and measure the execution time. Note that in our evaluation the graph algorithms include read-only random accesses to XLFDDs. As a result, the execution times of BFS, BC and SSSP using XLFDD are 88% to 141% of in-memory. If the required IOPS exceeds what XLFDDs can provide, or if there is data locality such that the CPU cache works effectively, DRAM is more advantageous. Otherwise, XLFDD can run faster than in-memory, which implies that, on average, the overhead to access the flash memories is less than the stall time for DRAM access. Also, the execution time using XLFDD scales as expected up to the range beyond the capacity of DRAM. Thus, XLFDD enables BFS-like processing of large graph inputs that don't fit into the DRAM capacity with execution times within 141% of in-memory, and in some cases faster than in-memory. Furthermore, we show that hiding access latency with stackless coroutines is effective not only for low latency flash memory but also for SCM and even DRAM. By hiding the latency with low CPU overhead, we show that even if the memory latency increases from tens of nanoseconds to several microseconds, the application performance does not degrade significantly as long as the IOPS are sufficient. We believe these results are the first example of how microsecond-latency memory can be used to achieve performance comparable to or better than DRAM on workloads with many small-sized random read accesses.

2 BACKGROUND

2.1 Flash memory and SSD

Flash memory is a memory element that composes SSD. A *die* of flash memory is composed of multiple *planes*, each plane is composed of multiple *blocks*, each block is composed of multiple *pages*, and each page is composed of memory cells. Reading from flash memory is done in page units, and the page size is usually a few KB to tens of KB. The time required to read one page is denoted *tR*. In normal flash memory, tR is tens of microseconds. In recent years, low latency flash memories have been announced [12, 15]. For example, in XL-FLASH [15], tR is shorter than 5 μs. SSD integrates multiple packages of flash memory and a controller. A package usually contains multiple dies. In order to achieve high performance, many outstanding requests are interleaved to the multiple dies in parallel. The random access performance is about 1 MIOPS for the latest enterprise high-performance SSD [16].

2.2 Load access or DMA access

Generally, DRAM is accessed by load instructions, while disk such as SSD is accessed by DMA (Direct Memory Access). However, for devices such as SCM or low latency flash memory, which has a latency between DRAM and SSD, it is hard to determine which access method is better in general. The load instruction can request data with only single instruction, but CPU execution may be stalled while accessing memory. The access latency can be hidden by the cache mechanism in the CPU or out-of-order execution. However, it is difficult to completely hide latency of more than hundreds of nanoseconds by the out-of-order execution. Further hiding is possible by using the prefetch instruction appropriately, but if the memory latency is at the microsecond level, there may not be enough outstanding requests that CPU can issue [21]. On the other hand, in the case of DMA, for example, when reading data, the source data position, destination memory address, data size, etc. are specified and a read request is issued. The read execution and data transfer are performed independently of CPU execution. The CPU can use the read data after confirming the completion of the data transfer. Generally, DRAM is specified as the transfer destination, but in some processors, the transferred data can be placed in the CPU cache. In case of DDIO [4] in Intel® Xeon®, write back and write allocate strategy can be used for the written data incoming from PCIe® interface. This allows the CPU to acquire the read data fast from the CPU cache. The disadvantages

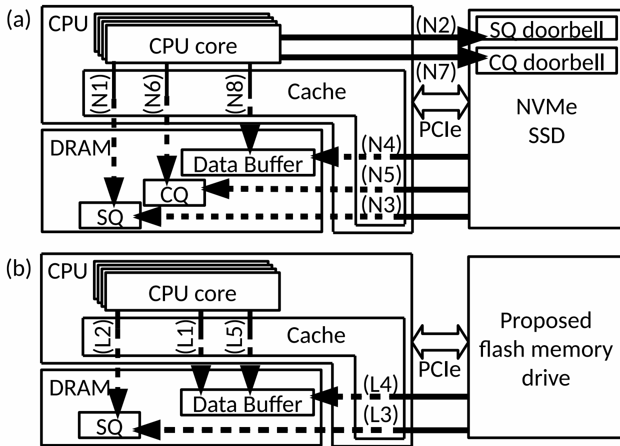

Figure 1: (a) Read operation of NVMe SSD. Access from the CPU or SSD to the DRAM is served faster in case of cache hit. Otherwise, the DRAM access depicted by the broken line occurs. (b) Read operation using the proposed lightweight interface. This is the case for checking the completion of the read operation by changing the value in the Data Buffer.

compared to the load instruction are that many instructions are required for issuing and completing requests. Furthermore, it is not possible to check at the hardware level if the data that is about to be read is already in the CPU cache. For devices such as SCM or low latency flash memory, DMA with a deep request queue can be effective to cover the long latency. However, it has been pointed out in [21] that handling the software queue by CPU incurs an overhead and the performance does not improve as expected.

2.3 NVMe interface

NVMe [8] is an efficient host interface for SSD. The NVMe interface is a DMA-based access protocol that runs on PCIe. Read operation of NVMe SSD is shown in Figure 1 (a). (N1) CPU writes a read request to SSD in Submission Queue (SQ). (N2) CPU writes a value to SQ doorbell register on SSD to notify that the request has been added to the SQ. (N3) SSD reads the SQ entry and (N4) executes the request command. The specified page in flash die is read and the read data is transferred to location specified in the request. (N5) SSD writes the completion of the request to Completion Queue (CQ). (N6) CPU detects that the entry in CQ has been added via an interrupt (or polling the CQ) and reads the CQ entry to know which request has been completed. Note that the CPU needs to read the CQ entry because the order of entries in the SQ and CQ can be different due to the internal processing of SSD. (N7) CPU writes a value to CQ doorbell register to notify SSD that the CQ entry has been consumed. (N8) CPU use the read data.

NVMe specification supports a deep SQ to drive many internal dies in parallel. The tR of a standard flash memory is tens of microseconds, but since multiple issued requests can be processed in parallel, large IOPS can be achieved. NVMe protocol is lightweight, but it takes about 100 ns CPU time per request [14]. As pointed out in [14], writing to the doorbell register is a large overhead. The doorbell is a register in PCIe MMIO space, and the write from CPU to the doorbell causes an access to uncached area. Furthermore, the memory fence instruction is inserted in order to ensure that N2 is executed later than N1, which can incur another large overhead.

3 PROPOSED ACCESS METHOD

3.1 Lightweight interface

We propose a new interface to reduce CPU overhead. The main strategies to reduce the overhead involved in NVMe protocol are the removal of doorbell and the improvement of the cache hit rate.

On the SQ side, flash memory drive polls the SQ entry to eliminate writes to the doorbell register in N2. The polling (referred to *SQ polling* in this paper) is done by PCIe read transactions from the drive to the SQ in DRAM. Periodic polling allows the drive to detect SQ entries without the doorbell write. The polling operation consumes PCIe bandwidth to some extent, but the performance impact is not significant, especially for read intensive workloads. Because PCIe connection is full-duplex, reading SQ entry and writing data from flash memory drive are in opposite directions, and there is little conflict for PCIe traffic. Even if the impact of the polling on application performance may be small, SQ polling is still not energy efficient and should be reduced if possible. Instead of always polling, instructions from the host to the device can be used to reduce unneeded polling. In our implementation, the CPU can specify the polling interval. Also, it can stop polling when there is no read request. By using a cache coherent protocol such as CXL [2] instead of PCIe in the near future, the drive may be able to detect the write to the SQ entry by the cache snooping message without polling. As a further modification regarding SQ, some fields in the SQ entry can be deleted to reduce the used CPU cache area and the CPU instructions for preparing the entry. In our implementation, the number of required fields is reduced by informing the drive in advance about fixed values that do not change on a per-request basis, instead of having a field for each entry.

On the CQ side, the queue structure itself is completely eliminated. This not only reduces doorbell write, but also contributes to improving the cache hit rate by eliminating access to CQ entries. The completion is notified by writing it to the location specified by the corresponding to each SQ entry, rather than being written into CQ. The CPU can detect the completion notification by polling (referred to *CN polling* in this paper) the memory location corresponding to each request. The CPU can suppress the number of CN polling by doing it after a sufficient amount of time has passed since the request was issued.

The memory location where the completion notification is written can have several options. In order to improve cache usage, the completion entry can be written at a position contiguous with the read data in Data Buffer. Note that the area for the completion entry in Data Buffer will be overwritten. To avoid it, the completion entry can be written at another location, for example, in the field in the corresponding SQ entry. Furthermore, the completion entry itself can be completely eliminated in the case of a read request. The CPU can detect whether the read operation is complete by polling the location in Data Buffer where the read data is placed. If part of the bytes in the Data Buffer have changed from the prewritten bytes, it means that read data has been delivered and the read

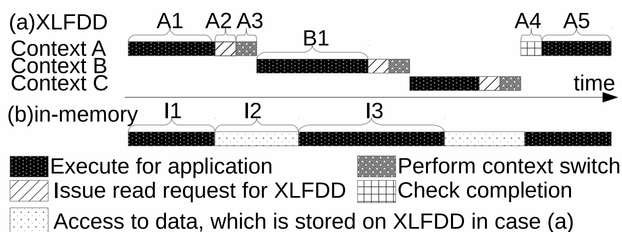

Figure 2: (a) Execution on one core with XLFDD and context switch. (b) In-memory execution on one core highlighting the access to data, which is stored on XLFDD in case (a).

operation is complete. This sequence can only be used if it is guaranteed that the prewritten data are different from the data to be read (e.g., set some bits in read data that are not used by the application as fixed values). Otherwise, the completion entry must be written explicitly. We have implemented all of the above completion notifications and allow users to choose. We use completion by the change of the prewritten bytes for graph processing in Section 4. When an exception such as a read error occurs, it can be notified separately from completion through an interrupt to the CPU, which has not yet been implemented in our experiments.

The flow of a read operation using proposed lightweight interface is shown in Figure 1 (b). This is the sequence for checking the completion of the read execution by changing the value in the Data Buffer. (L1) CPU writes different data to Data Buffer from the data transferred from flash memory drive. (L2) CPU writes a request to drive in SQ. Note that the SQ entry will be reusable as soon as the drive retrieves it in our evaluation. Therefore, if the depth of the SQ is larger than the number of outstanding requests, new requests will not overwrite requests that the drive has not yet retrieved. (L3) Drive polls the SQ entry (SQ polling). The polling may have been being done since before L1. (L4) Drive executes the request command. The specified page in flash die is read and the read data is transferred to location specified by the request. (L5) CPU polls Data Buffer (CN polling). If the data written to Data Buffer in L1 has changed, the read operation is completed and the data in the Data Buffer can be used. If not, CPU polls later again.

The proposed interface is lightweight due to the above modifications. However, in order to achieve higher performance using the interface, it is still important that the rate of cache hit for accesses to SQ or Data Buffer is improved as much as possible. In addition, L5 should be performed once at the appropriate timing to avoid unnecessary CN polling. These are related to the number of contexts (described in detail later) in application level. Its experimental results are shown in Section 5.3.2.

Hardware implementation. We have implemented the proposed interface on prototype board, XLFDD [46]. The XLFDD has a 2.5-inch form factor equipped with eight XL-FLASH packages and an FPGA. We have implemented FPGA logic so that the XLFDD can act as a prototype of a flash memory drive such as an SSD with limited functionality. The FPGA handles the proposed lightweight interface, controls the XL-FLASH dies, and corrects errors in the data read from the dies. XLFDD communicates with the host via PCIe Gen3 4 lanes. The minimum read size of XLFDD is 16 bytes, while that of a general NVMe SSD is 512 bytes. The page size of XL-FLASH die is 4KB. For small-sized read, not full page but the required part of data is transferred to the FPGA controller and host, which reduces cache pollution in the CPU. XLFDD is accessed by specifying a physical address from the host. Therefore, software running on the CPU is responsible for conversion from a logical address to a physical address. While tR of XL-FLASH is in less than 5 μs, the read latency of single 64-byte read from CPU is about 9 μs. By reading multiple dies in parallel, the peak performance of one board reaches 11 MIOPS when reading less than 64 bytes. Although the details are not discussed in this paper, write access is possible as well. In addition, the conventional doorbell access without SQ polling is also implemented. It may be possible to use the proposed interface together with NVMe commands using a different SQ, although we have not implemented it yet.

Here's a short summary of this section: doorbell is gone, SQ now relies on SQ polling, and completion is no longer notified by queue. Also, the cache hit rate is improved by the following factors (1) Field reduction in SQ (2) No CQ (3) Only necessary data is sent to Data Buffer. (4) Selection of appropriate number of contexts.

3.2 Hiding latency using stackless coroutines

Although the latency of reading from XLFDD is several microseconds, the CPU can overlap other processes during read operation due to DMA-based access. In order to achieve high IOPS performance, it is necessary to issue thousands of outstanding requests and operate many dies in parallel. To realize this, hundreds of contexts operate in parallel and are switched on one CPU core, as shown in Figure 2 (a). (A1) While executing context A, (A2) a read request is issued when it needs to access the data on XLFDD. (A3) Context switch is performed, and (B1) another context B is processed on the core. During the processing of context B, a read request to XLFDD is issued if necessary, and another context switch is executed in the same manner as context A. (A4) When execution returns to context A, it is checked if the read request is completed, and (A5) processing of context A restarts if completed. Otherwise, switch to another context is performed again. In Figure 2 (a), only three contexts are shown, but hundreds of contexts are used per core to hide several-microsecond latency.

Although the latency can be hidden by the method of Figure 2 (a), it is important to reduce the overhead such as A2, A3 and A4. For comparison, Figure 2 (b) shows an example of in-memory execution. All data is located on DRAM, including the data located on XLFDD in case of Figure 2 (a). Note that the in-memory execution can be executed without context switch. I1 is the same as A1. I2 highlights the access to the data on DRAM, which is stored on XLFDD in case of Figure 2 (a). If this is a random access to a large memory space, the cache in the CPU may not hit, and thus load access to the DRAM occurs. Although typical DRAM access latency is about 90 ns [25], the CPU stall cycles for the load access can be shorter than 90 ns since the latency is partially or completely hidden by out-of-order execution. On the other hand, in Figure 2 (a), the random access to DRAM is replaced by reading from XLFDD. As shown in Section 5.2, the total of A2 and A4 can be reduced to about 6.7 ns by using the new interface proposed in Section 3.1.

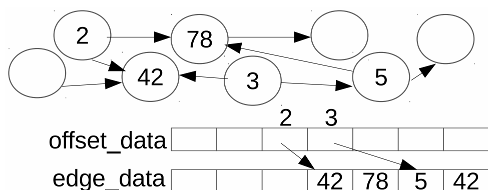

Figure 3: An example of a graph expressed in CSR format.

The remaining overhead A3 is the cost of context switch. A conventional context switch has a large overhead, taking several microseconds for OS-based ones [34] and several tens of nanoseconds for user-space ones [21]. We will reduce it to a few nanoseconds by using *stackless coroutines*.

A coroutine can describe the behavior of suspending a function and resuming the suspended function after executing another function [26]. The coroutine is classified into stackful coroutine and stackless coroutine. In a stackful coroutine, each execution context has a stack, and, the register values in the CPU are saved in the stack when the context is switched. On the other hand, stackless coroutine does not have stack for each context. Therefore, it is necessary for the user to program the saving of data that will be needed later when the execution is suspended. If the amount of data to be saved is small, fast context switch is possible. In the best case, only the program counter can be changed. The stackless coroutine is standardized in C++20 and is expected to be widely used in the near future.

By using the stackless coroutine, a load instruction in in-memory execution can be replaced with a read access to XLFDD as shown in Listing 1. `read_req` performs L1 and L2, and `check_completion` performs L5 in Figure 1 (b), which are user-space library functions. `context_id` is used to identify the Data Buffer corresponding to the coroutine context. Note that there is no performance degradation by checking completion when there is no read request since `check_completion` is only executed after `read_req`.

Listing 1: Access to XLFDD with coroutine

```
XLFDD.read_req(context_id, read_address, size)
do {
  yield() // suspend
} while (XLFDD.check_completion(context_id))
```

Coroutine makes it easy to implement XLFDD applications with hiding latency like [26]. In general, it is possible to achieve large IOPS without using coroutine, but the implementation can be complicated and difficult to understand. By using coroutines, there is little changes from the original algorithm and the structure can be kept easy to understand. Note that it is currently necessary to manually partition the data into a fast tier (DRAM) and a slow tier (XLFDD), and optimization of the partitioning is not the focus of this paper. In the following sections, we apply the stackless coroutines to actual workloads and evaluate the performance in application level.

4 APPLICATION : LARGE-SCALE GRAPH PROCESSING

4.1 In-memory graph processing

Applying the access method proposed in Section 3 to graph processing, we partially replace DRAM with XLFDD in this section. Large-scale graph processing is an application in which small-sized random access frequently occurs to a large dataset. The CSR (Compressed Sparse Row) format is a data structure often used to represent graph structure. Figure 3 shows an example of a graph represented in CSR format. The CSR format consists of two arrays, `offset_data` and `edge_data`. Each element of the `offset_data` corresponds to a vertex and is a pointer to the start position in the `edge_data` where the edge data of that vertex is stored. The `edge_data` holds all edge data, and the contents are stored for each vertex. In order to obtain the edge information connected to a certain vertex, the element of the `offset_data` is obtained with the vertex identifier as the index, and the elements of the `edge_data` is obtained by dereferencing the pointer. In graph processing, especially BFS-like algorithms, when processing the adjacent vertices by traversing the edges connected to active vertices (called *frontier* later) on the graph, access patterns to both `offset_data` and `edge_data` tend to be random. Note that in this paper, the focus is on static graphs, so access to the `offset_data` and `edge_data` is read only in the processing after graph initialization.

Several in-memory graph processing frameworks have been proposed for fast graph processing [18, 41, 47]. However, in large-scale graphs, there are cases where the `offset_data` and `edge_data` are too large to fit in DRAM. Since the `edge_data` is generally several to several tens of times larger than the `offset_data`, we put the `edge_data` outside the DRAM, that is, in the XLFDD in our evaluation. In this paper, we use the GAP benchmark suite [18] as a baseline for implementing graph processing. GAP is a benchmark suite that includes several in-memory parallel graph algorithms. It is implemented in C++ and uses OpenMP for parallelization. We choose GAP because it is a pure C++ implementation rather than a DSL (Domain Specific Language), which can be easily rewritten for XLFDD, and it is one of highly optimized implementations.

4.2 Replacing DRAM with XLFDD

Listing 2 is a pseudo code abstracted from implementation of BFS-like graph algorithms in GAP.

Listing 2: Abstracted code of original GAP implementations

```
X1: #pragma omp parallel for schedule(dynamic, 64)
X2: foreach u in frontier
X3:   offset = offset_data[u]
X4:   deg = offset_data[u+1] - offset_data[u]
X5:   foreach i in [0, deg]
X6:     v = edge_data[offset + i]
X7:     do_calculations(v, u, ...)
```

The `frontier` of Line X2 holds the set of vertices that are currently active. For each vertex `u` in the `frontier`, `offset_data` and `edge_data` are accessed, and each `v` that is an adjacent vertex of `u` is enumerated. Line X7 updates the data for vertex `v` depending on the actual graph processing (e.g. updates visiting state in BFS).

Usually, the next `frontier` is also updated in Line X7. In GAP, the for-loop in Line X2 is parallelized by OpenMP and executed by multiple threads.

When using XLFDD for the application code in Listing 2, it is necessary to rewrite the code as in Listing 3. Here, `edge_data` is placed on the XLFDD. In the pseudo code, `sched_coro` starts hundreds of coroutines per core, resumes coroutine suspended by `yield()`, and repeats until `return()`.

Listing 3: Modified code for XLFDD

```
Y1  : coro_func(sub_frontier, page_idx) {
Y2  :   if sub_frontier.empty()
Y3  :     if frontier.empty()
Y4  :       return()
Y5  :     sub_frontier = get_next(frontier, N)
Y6  :   u = sub_frontier.pop()
Y7  :   offset = offset_data[u]
Y8  :   deg = offset_data[u+1] - offset_data[u]
Y9  :   edge_size = conv_to_edge_size(deg)
Y10 :   XLFDD.read_req(ctx_id, offset, edge_size)
Y11 :   // save data if necessary such as u or deg
Y12 :   do {
Y13 :     yield()
Y14 :   } while (XLFDD.check_completion(ctx_id))
Y15 :   // restore the saved data
Y16 :   foreach i in [0, deg]
Y17 :     v = data_buffer[ctx_id][i]
Y18 :     do_calculations(v, u, ...)
Y19 : }
Y20 :
Y21 : #pragma omp parallel // for each core
Y22 : foreach j in [0, num_core]
Y23 :   sub_frontier = [], page_idx = 0
Y24 :   sched_coro(coro_func, sub_frontier, page_idx)
```

Y7 to Y18 is the main part corresponding to X3 to X7 in Listing 2. In the CSR format, the data of the vertices adjacent to the vertex `u` are placed in a contiguous area, so they can be obtained with one read request as long as it fits in one page. Therefore, instead of replacing the access to `edge_data` of Line X6 with the access to XLFDD every time, we access XLFDD once outside the loop of Line X5. A `data_buffer` is statically prepared for each context, and the data read from XLFDD is transferred to the `data_buffer` by DMA. Note that the total size of `data_buffer` is several MBytes.

The loop annotated by OpenMP macro in Line X2 is processed in parallel by combining multithreads and coroutines. In GAP implementation, a directive `schedule(dynamic, 64)` is specified for load balancing among threads. This means that each thread will process the every specified number of entries, here 64. A thread that has finished processing 64 entries acquires 64 new ones from `frontier` and processes them. Similarly, we obtain the specified N entries for each thread from `frontier` (Y5). The acquired entry group is `sub_frontier`. In the `sub_frontier`, each coroutine sequentially acquires and processes vertices from the front (Y6). This is the best of all we have tried with parallelization strategies. Note that threads are created for the number of cores to be used, and each thread is fixed to the corresponding core in our evaluation.

4.3 Implementation details

4.3.1 Physical addressing. As shown in Section 3.1, XLFDD is accessed by specifying a physical address. By storing the physical address location of the corresponding edge data in `offset_data`, it is possible to issue a request to the edge data without converting the logical to physical address every time it is accessed. The physical address in XLFDD is not a linear address space because there may be unusable memory blocks in the middle like missing teeth. The `edge_data` should be placed allowing gaps to avoid the unusable memory blocks. However, in this case, the number of edge data of a certain vertex `u` (called *degree*) cannot be calculated by the difference between the physical addresses of `u` and `u+1` like Line Y8. Instead, the degree is stored separately in the array `deg_data` on DRAM. By placing `offset_data[u]` and `deg_data[u]` on the same cache line, access overhead to `deg_data[u]` can be negligible. Note that `deg_data` may consume additional DRAM space, but is an order of magnitude smaller than `edge_data`.

4.3.2 Page boundary. In general, most of the edge data per vertex is much smaller than the page size of flash memory. Also, as described above, it is possible to place `edge_data` in non-contiguous physical memory address spaces. Therefore, by arranging the edge data per vertex so as not to span multiple pages as much as possible, most of the access to edge data per vertex can be done by reading one page from XLFDD. If the size of edge data per vertex exceeds one page, it is necessary to read multiple pages. When dealing with a graph in which the distribution of degrees follows power-law as shown in Figure 4 (b) and (c), there are few vertices with very large edge data size. We handle the large edge data in multiple coroutine contexts. Listing 4 shows the final version of the code snippet that replaces Y6 to Y18 part of Listing 3.

Listing 4: Final version of code for XLFDD (to replace Y6 to Y18 with Z1 to Z15)

```
Z1 : u = sub_frontier.first()
Z2 : offset = offset_data[u]
Z3 : deg = deg_data[u]
Z4 : edge_size = conv_to_edge_size(deg)
Z5 : if edge_size < 1page
Z6 :   XLFDD.read_req(ctx_id, offset, edge_size)
Z7 :   /* do Y11 to Y18 */
Z8 :   sub_frontier.pop()
Z9 : else
Z10:   XLFDD.read_req(ctx_id, offset+page_idx, 1page)
Z11:   page_idx += offset_for_1page
Z12:   /* do Y11 to Y18 */
Z13:   if (all edge data of u are consumed)
Z14:     sub_frontier.pop()
Z15:     page_idx = 0
```

5 EVALUATION

5.1 Experimental Setup

We focus on the following BFS-like graph algorithms in our evaluation. The caching mechanism tends to be less effective for these algorithms due to the poor locality. Note that how much the actual access pattern has locality depends on the input graph.

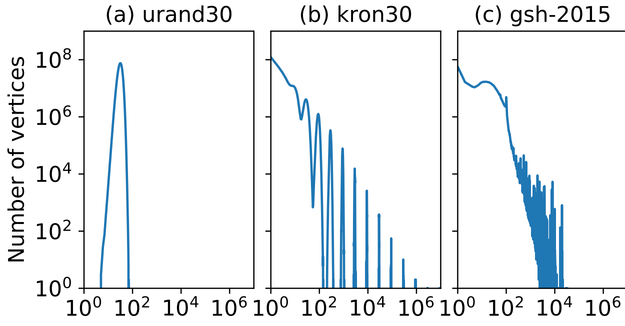

Figure 4: The degree distributions of urand30, kron30 and gsh-2015. The horizontal axis is degree.

Table 1: Graph datasets used for evaluation. (B stands for billion in # of vertices/edges.)

	# of vertices	# of edges	edge data size
urand30	1B	32B	128GB
kron30	1B	32B	128GB
sk-2005	51M	1.9B	7.8GB
twitter-2010	42M	1.5B	5.9GB
gsh-2015	988M	33B	135GB
wdc12	3.6B	128B	1TB

Breadth First Search (BFS) repeats updating the active vertex set by visiting unvisited vertices adjacent to each vertex in the currently active vertex set. The active vertex set starts from a (randomly selected) vertex and repeats this process until all the reachable vertices are visited. GAP optimizes switching between top-down and bottom-up BFS [17]. However, we focus on evaluating random access performance in this paper, so we always use top-down BFS. OpenMP parallelization of GAP top-down BFS has no `schedule` directive. Adding `schedule` directive allows appropriate load balancing among threads and enables faster execution even if it is in-memory execution. Therefore, we add `schedule(dynamic, 64)` for top-down BFS.

Betweenness Centrality (BC) counts the shortest path through each vertex. After performing a top-down BFS starting from a vertex as a forward processing, each vertex on the graph is visited in reverse order and the amount of contribution is accumulated according to the number of shortest paths through that vertex as a backward processing. In our evaluation, forward and backward processing are performed once from a random starting point.

Single Source Shortest Path (SSSP) finds the shortest path from a starting point to each vertex in weighted graph. The weighted edge is stored as a pair of adjacent vertex data and its weight as edge data. Therefore, the size of `edge_data` is twice as large as that of the unweighted graph. GAP implements a delta-stepping algorithm suitable for multithreading. The parameter Δ for delta-stepping does not significantly affect performance in our experiments, so we keep the default value $\Delta = 1$.

The graph datasets used for evaluation are shown in Table 1. Note that in GAP, if the number of vertex is more than 2^{31}, the data size doubles because it uses 64-bits per vertex instead of 32-bits. There are synthetic graphs and real-world graphs. Synthetic graphs are uniform random graphs (*urand*) and Kronecker graphs (*kron*) [33] implemented in GAP. The parameters of kron follows Graph500 [3] specifications. The log of the number of vertices is called *scale* in synthetic graphs. For example, if scale is 26, the graph has 64M vertices. In this paper, urand graph with scale 26 is referred to as urand26. The average degree of all synthetic graphs is fixed to 32 in our experiments. As shown in Figure 4 (a), the degree distribution of urand is narrow, centered on 32. On the other hand, kron follows power-law distribution. In general, kron has a distribution of degrees similar to those of real-world graphs compared to urand. The Table 1 shows the case of scale 30 as an example, but our evaluation covers up to scale 32. As the real-world graphs, we use sk-2005, twitter-2010 and gsh-2015 in the webgraph dataset [10, 19, 20] and wdc12 [1]. As an example of the degree distribution of the real-world graph, that of gsh-2015 is shown in Figure 4 (c). The evaluated real-world graphs have power-law or power-law-like distribution. There are a large number of small degree vertices smaller than 100 bytes, but also a few very large degrees. Note that the real-world graphs are directed while the synthetic graphs are undirected in our experiments.

We conduct all experiments on a non-uniform memory architecture machine with two Xeon Gold 5218 processors running at 2.3 GHz. Each processor has 16 physical cores and 6 memory channels, each of which is connected to 32 GB DRAM of DDR4-2933 and Optane DCPMM 128 GB. Nine XLFDDs are connected to one processor via a 16-lane PCIe switch. Note that 16 lanes have sufficient bandwidth for large IOPS accesses smaller than 100 bytes. At the time of execution, only the processor and the memory connected to XLFDDs are used by the affinity setting and `numactl` command unless otherwise noted. It runs with 16 cores. Ubuntu 18.04.4 and kernel 5.3.0 are used. As the compile option, `g++ -std=c++11 -O3 -march=native` is used.

We compare the performances by placing `edge_data` on DRAM, XLFDD, and DCPMM. DCPMM is used in AppDirect Mode to access the `edge_data` through dev-dax unless otherwise noted. In all three configurations, large arrays such as `offset_data` are allocated on hugepages in heap. It reduces TLB miss and contributes to performance improvement in all configurations.

5.2 CPU overhead per request

In order to measure the actual CPU overhead of making requests with the lightweight interface, we perform simple random read accesses to XLFDDs. Since nine XLFDDs are used, each providing up to 11 MIOPS, the peak potential performance is 99 MIOPS. Table 2 (a) shows the performance result when read requests are issued to nine XLFDDs using a simple loop by single CPU core. Note that the loop can be realized with only a few instructions including compare and branch, and the CPU overhead for controlling loop is negligibly small. The requests are ideally issued for all the dies without biasing to a specific die. By using the lightweight interface, it is possible to issue read requests at a sufficient speed even if only one CPU core is used, and it is possible to achieve 99 MIOPS. This means that the CPU core can process each request in less than 10.1 ns. Since the result is limited by the IOPS upper limit of XLFDD, the net CPU processing time per request is not accurately measured. Therefore, as a further measurement, by

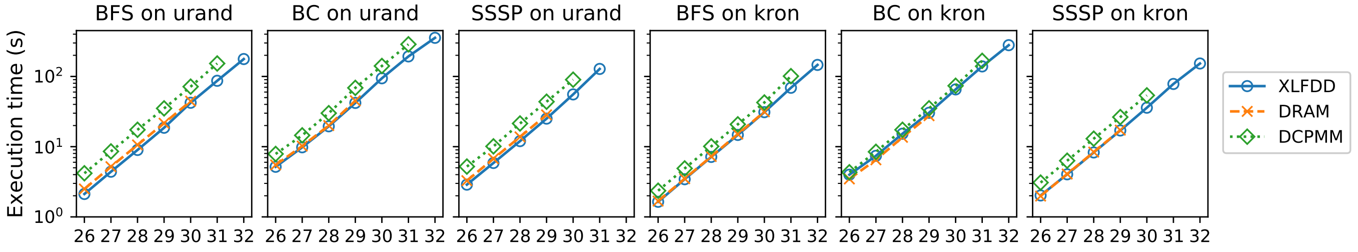

Figure 5: The results of execution time of graph algorithms on synthetic graphs. The horizontal axis is the scale of the graph.

Table 2: Performance results when read requests are issued without biasing to a specific die using one CPU core. CPU overhead per request is calculated by the reciprocal of IOPS.

	IOPS	CPU overhead per request
(a) 9 XLFDDs	99M	10.1 ns
(b) 9 XLFDDs without waiting tR	150M	6.7 ns
(c) NVMe SSDs	12.1M	82 ns
(d) 9 XLFDDs + stackless coroutines	99M	10.1 ns
(e) 9 XLFDDs + stackless coroutines without waiting tR	150M	6.7 ns

Table 3: The execution time and its ratio to DRAM case at the largest comparable scale where all data fits in DRAM in Figure 5.

	Dataset	Execution time (s) and its ratio to DRAM case		
		XLFDD	DRAM	DCPMM
BFS	urand30	42.2 (0.95)	44.5 (1.00)	71.6 (1.61)
BFS	kron30	30.8 (0.98)	31.4 (1.00)	42.6 (1.36)
BC	urand29	42.0 (0.93)	45.1 (1.00)	68.8 (1.52)
BC	kron29	30.4 (1.10)	27.6 (1.00)	35.2 (1.27)
SSSP	urand29	25.1 (0.88)	28.4 (1.00)	43.7 (1.54)
SSSP	kron29	17.0 (0.99)	17.2 (1.00)	26.4 (1.53)

returning the data from the buffer in die without waiting for tR, the CPU processing time per request is measured under the condition that IOPS is not the bottleneck. Table 2 (b) shows the results. 150 MIOPS with single core is obtained, that is, each request can be processed in about 6.7 ns, which is less than 1/10 of CPU overhead for NVMe interface. In our environment, NVMe SSD requires 82 ns per request as shown in Table 2 (c). This value is calculated from the IOPS when a total of 12 NVMe SSDs are used. Optane Memory [11] and Optane 800P [13] are used as NVMe SSDs. Since they provide about 1.4 MIOPS at 512B random access with a single unit, 12 units are capable of providing 16.8 MIOPS. However, it is limited to 12.1 MIOPS because of the CPU overhead.

Table 2 (d) and (e) show the results when a sufficiently large number of contexts are operated using stackless coroutines. In this case, a read request and its completion is handled by the same context as shown in Listing 1. Also, a read request is not issued until the completion for the previous request is confirmed in a context. Hence, the maximum number of outstanding requests is the number of contexts. In the normal case of waiting for tR, 99 MIOPS is obtained, and in the special mode of not waiting for tR, 150 MIOPS is obtained, by single core. As mentioned in Section 3.2 stackless coroutine context switches have a small overhead, which has also been experimentally shown as (d) and (e) can achieve the similar performance as (a) and (b).

5.3 Evaluation results for synthetic graphs

Figure 5 shows the execution time of graph algorithms with 16 threads on urand and kron. There are some scales that cannot be executed due to insufficient memory capacity. Since the time complexity of these algorithms is proportional to the number of vertices or edges, it is expected that if the scale is increased by 1, the number of vertices or edges to be processed will double and the execution time will also double. The results in Figure 5 are roughly in line as expected.

The Table 3 also shows the execution time and its ratio to DRAM case at the largest comparable scale where all data fits in DRAM in Figure 5. When XLFDD is used, the algorithms can be executed in 88% to 110% of the in-memory execution time. It implies that read access to XLFDD and context switch is faster than random access to DRAM in some cases. The execution time of DCPMM case is about 127% to 161% of that of DRAM. Simply replacing DRAMs with DCPMM can result in performance degradation. According to the *cycle_activity.stalls_mem_any* counter in Linux `perf` command, the increase in execution time of DCPMM is due to the increase in stalls for memory access. Waiting for random access to the DCPMM is likely the reason for the increased execution time. Software optimizations may be needed to hide the access latency.

5.3.1 IOPS performance. If IOPS performance provided by XLFDD is sufficient, XLFDD will deliver good performance. When comparing urand with kron, the performance ratio of XLFDD to in-memory is better for urand. This is mainly because the required peak IOPS of kron is higher than that of urand, and nine XLFDDs are not sufficient for the peak IOPS of kron. The required IOPS performance depends on the ratio of IO to computation. Also, there can be a time variation of the required IOPS within an execution. In urand, the degree is almost around 32, which means that edge data per vertex is always read in units of around 128 bytes. On the other hand, in kron, a large number of 4-byte accesses occur. After reading 4 bytes of data, the CPU time required to process the data is shorter than that of 128 bytes, and the next read request for the next edge data is issued relatively early. As shown below, the

Table 4: Results of BFS on kron30 using DRAM and using XLFDD while changing the number of contexts.

	XLFDD								DRAM
# of contexts per thread	1	2	16	32	64	128	256	512	N/A
Execution time (s)	321	168	42.4	35.4	31.7	30.8	32.2	35.1	31.4
LLC-load miss rate	72.3%	72.3%	72.4%	72.8%	73.8%	75.1%	76.3%	77.4%	75.8%
DTLB-load miss rate	0.03%	0.16%	1.5%	2.5%	3.6%	4.9%	5.8%	6.9%	6.0%
# of `yield()` ($\times 10^9$)	8400	710	70	27	15	7.0	3.5	0.49	N/A

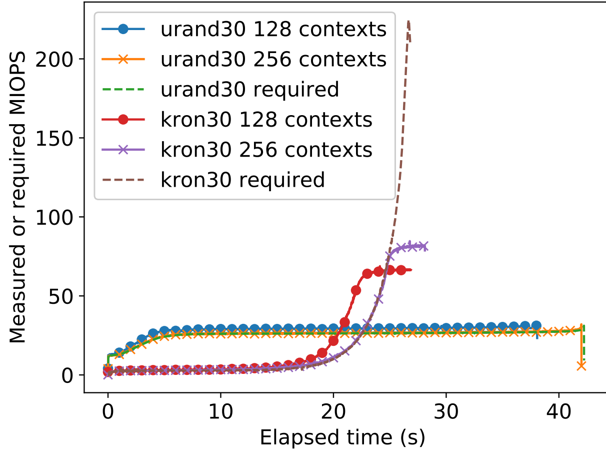

Figure 6: The measured IOPS values when BFS is executed on urand30 and kron30. *128 and 256 contexts* mean the number of coroutine contexts per thread in XLFDD case, respectively. *Required* shows IOPS requirement estimated from in-memory execution results.

required IOPS is likely to be high during the period when 4-byte reads occur frequently in kron, and the IOPS provided by XLFDDs may be insufficient in such period.

Figure 6 shows the measured IOPS values when BFS is executed on urand30 and kron30. The solid lines show the total IOPS value of nine XLFDDs actually measured. Figure 6 also shows the estimated IOPS value required to achieve the same speed as when executing in-memory by the dashed line. For urand30, the required IOPS is almost constant at about 30 MIOPS, and XLFDD provides sufficient IOPS. Using XLFDD with 128 contexts per thread is faster than in-memory, which implies that the CPU overhead for read accesses is shorter than DRAM stall cycles. For kron30, XLFDD is faster than in-memory up to about 25 seconds after the start. During this period, IOPS provided XLFDDs are sufficient. Especially in the range of tens of MIOPS, it can be seen that access to XLFDD has a substantial advantage over frequent DRAM accesses. In the final iterations of kron30, the required IOPS has increased significantly, eventually reaching over 200 MIOPS. This is because the ratio of IO to computation tends to be large as below. In the latter half of BFS, vertices with a small degree are likely to remain unvisited. Furthermore, the number of the unvisited vertices within the acquired edge data is small, which reduces the amount of CPU processing. In the final step, XLFDD with 128 contexts per thread peaks at around 66 MIOPS. Therefore, in-memory execution

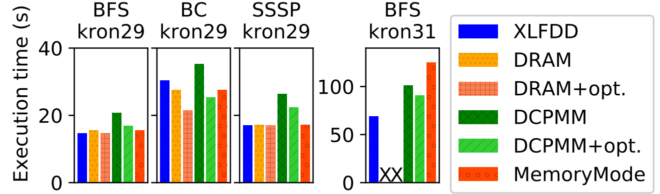

Figure 7: Comparison on kron29 and kron31. *+opt.* denotes applying the optimization using prefetch instruction and `yield()`. X denotes Out Of Memory.

is faster in the final iterations. As the whole BFS, using XLFDD is slightly faster than in-memory.

As described in Section 5.2, peak performance of nine XLFDDs is 99 MIOPS. The main reason why only 66 MIOPS can be obtained here is that the number of requests are not sufficient to demonstrate 99 MIOPS. The access pattern is different from the ideal case in Table 2. If the number of requests is not enough, not all dies will work since the request may be slightly biased to some dies. By using 256 contexts per thread, the measured IOPS is improved up to 82 MIOPS, but BFS execution time is longer than when using 128 contexts. If the number of contexts is increased too much, the performance degrades in our evaluation as described in Section 5.3.2.

5.3.2 Impact of the number of contexts. We show that good performance can be obtained by selecting an appropriate number of coroutine contexts. Table 4 shows the execution time, the LLC-load miss rate, the DTLB-load miss rate, and the number of `yield()` when executing BFS on kron30 using DRAM and using XLFDD with 1, 2 and 16 to 512 contexts per thread. LLC-load and DTLB-load miss rate are measured using `perf` command. According to Table 4, the smaller the number of contexts, the lower the LLC-load and DTLB-load miss rate. Since contexts are operated on CPU core concurrently, if there are many contexts, they can compete caches and TLBs causing more misses. On the other hand, as the number of contexts increases, the number of `yield()` decreases. When the number of contexts is large, it takes a long time to return to the own context after suspending all other contexts. It tends to be the case that the read completion has already arrived when returning to its own context. By choosing the appropriate number of contexts, we can choose the optimal point for the trade-off between cache contention and the number of `yield()`. In the case of Table 4, 128 contexts is the best. Although the optimal number of contexts varies depending on the algorithm and input graph, it lies between 64 and 256 for the range of workload we have experimented with. Basically, 256 contexts are suitable when there are

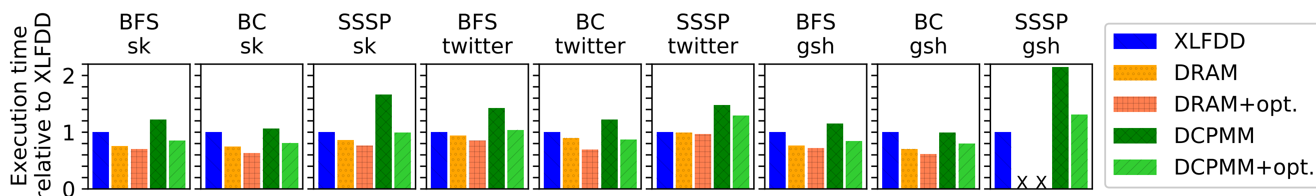

Figure 8: The results on the real-world graphs. The vertical axis is the execution time relative to the execution time using XLFDD. sk-2005, twitter-2010 and gsh-2015 are denoted by sk, twitter and gsh, respectively. `X` in the figure denotes that it could not be executed due to Out Of Memory. *+opt.* denotes applying the optimization in Section 5.3.3.

many small-sized accesses such as 4 bytes, and 128 or 64 contexts are suitable when many accesses are several tens of bytes or more. It is a future work to adaptively change the number of contexts during graph processing.

One of the important points to note in Table 4 is that the LLC-load and DTLB-load miss rate will be lower than when executing on DRAM if an appropriate number of contexts is selected. When accessing `edge_data`, the access by the load instruction is a random access to a huge memory space and tends to cause both LLC-load miss and DTLB-load miss. On the other hand, in the case of access by DMA, the data is accessed by the load instruction after transferred to the Data Buffer. The cache hit rate can be improved by accessing the transferred data before it is evicted from the cache. Also, since the size of the Data Buffer is relatively small, there are few DTLB misses caused by access to the edge data.

5.3.3 Hiding latency for DRAM/DCPMM. Context switching with stackless coroutines is so fast that it can also hide DCPMM and even DRAM access latency. Figure 7 shows the result of optimization using the prefetch instruction and `yield()` in a similar manner to Listing 4. In DCPMM+opt. case, CPU stall due to memory accesses has been reduced, and performance is approaching that of DRAM. Moreover, some DRAM cases can be improved by the optimization. This figure is the case for kron29, but the similar trend can be found for urand and other scales, (as well as in the real-world graphs in Section 5.4). By using fast context switches, memories with read latency of a few hundred nanoseconds or a few microseconds can bring the performance close to that of DRAM. Figure 7 also includes results using DCPMM Memory Mode, which uses DRAM as a cache for DCPMM. As shown in the result of kron29, as long as the cache hit occurs, the latency of DCPMM can be hidden and the performance is equivalent to that of DRAM. On the other hand, in kron31, DRAM cache misses often occur, resulting in poor performance. Due to space limitations we have not included results other than BFS on kron31, but we observed similar performance degradation for graphs larger than DRAM capacity.

5.3.4 Applying our method for sequential read accesses. For sequential reads, the existing interface such as NVMe interface is enough to read in large units and buffering them in DRAM as described in Section 1. However, if the IOPS is sufficient, our proposed method to issue small-sized accesses every time performs well even for sequential access since the CPU overhead for each access is very small. As a simple experiment, we use PageRank (PR) and Connected Components (CC), which contain a lot of sequential accesses to `edge_data`. We execute those in GAP benchmark with read accesses to `edge_data` on XLFDDs for each vertex `u` like Listing 4. The execution time is 1.17 times and 1.66 times slower than in-memory on kron26, respectively. In the case of PR, IOPS is sufficient even if read requests are issued for each vertex, but it is insufficient in CC. For the better performance, read requests should be issued in large units utilizing the characteristic of sequential access. It should be noted that our proposed interface can be used with read requests in large units, such as NVMe commands, although this has not yet been implemented.

5.4 Evaluation results for real-world graphs

Figure 8 shows the results on the real-world graphs. Each bar in the figure is the relative time to the execution time when XLFDD is used. According to Figure 8, some graph algorithms, especially on twitter-2010, can be executed in about 110% of in-memory execution time. That is, the performance close to that of in-memory execution is obtained when XLFDD is used. However, other results such as BFS on sk-2005 take up to 141% longer than in-memory execution. This is mainly because the real-world graphs have higher locality or lower randomness in the graph than synthetic graphs. In these cases, high locality means that adjacent vertices have close identifiers and are close together in memory.

In order to investigate the impact of data locality, the following experiment is conducted. For the graph data obtained from the webgraph dataset, the graph identifier is reordered at random while the graph structure is maintained. The results on those graphs are shown in Figure 9. The bars are also relative time to the execution time of XLFDD in Figure 8. When using XLFDD, there is not much difference in execution time between Figure 8 and Figure 9. On the other hand, when using DRAM or DCPMM, there are many cases in which the execution time increases due to lowered locality through the randomization. The execution result of Figure 9 is similar to that of the synthetic graph. It indicates that our proposed method achieves performance close to that of in-memory when there are many random accesses, while if there is some locality in data access, the current implementation using XLFDD is likely to be lower in performance than the one using DRAM. High locality favors in-memory over XLFDD for two factors. First, load access to the `edge_data` is likely to cause a CPU cache hit. When access to the `edge_data` is less random, XLFDD, in which a read request is always issued without checking whether the requested data is in the CPU cache, is disadvantageous. The other factor is that the required IOPS tends to be high, and the IOPS provided by

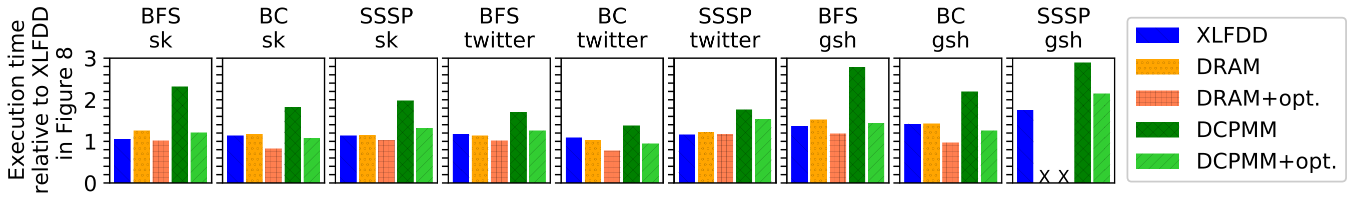

Figure 9: The results on the real-world graphs, whose vertices are randomly reordered. The vertical axis is the execution time relative to the execution time using XLFDD in Figure 8. The rest of the notation is the same as in Figure 8.

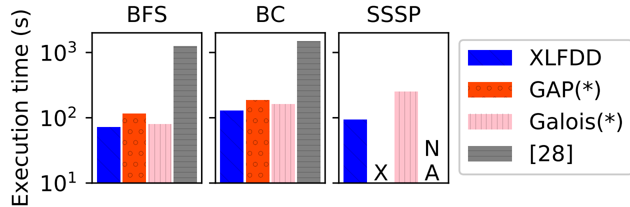

Figure 10: The results on wdc12. (*) is executed on our machine with 16 threads and 1.5TB memory in Memory Mode. X denotes Out Of Memory and NA denotes Not Available in [28].

XLFDD may be insufficient compared to DRAM case in a certain period. When locality increases, CPU cache hit rate of not only `edge_data` but also `offset_data` and other data on DRAM necessary for graph processing increases. These cache hits reduce computation time and increase the frequency of accessing `edge_data`. Therefore, the required IOPS tends to be high. Further optimizations in XLFDD case for applications that have some locality, such as utilizing the cache, is future work.

5.5 Comparison with existing works

There are existing works for processing graphs of sizes that do not fit in DRAM, such as [24] using SCM and [28] using flash memory. In [24], several in-memory graph processing is executed on Memory Mode of DCPMM, which allows large graphs to be treated as if they are on DRAM. However, for graphs where a large number of accesses to the DCPMM occur, performance may be degraded as shown in Section 5.3.3. We compare the existing work with XLFDD on a very large graph, wdc12, and the results are shown in Figure 10. We execute Galois [41], which appeared to be the fastest in [24], using DCPMM Memory Mode. Note that the capacity of the DCPMM of one processor is not enough to hold wdc12, so the DCPMM of both processors with a total capacity of 1.5TB is used. On our machine, Galois and GAP on Memory Mode are slower than XLFDD, suggesting that frequent accesses to DCPMM might cause performance degradation as shown in Section 5.3.3. [28] is a system using flash memory that can handle very large graphs by putting all the data, including vertex data as well as edge data, on flash memory. However, a large amount of reading and writing of flash memory occurs, which hinders the performance. Since [28] uses special hardware and we cannot experiment with it in our environment, the values reported in [28] are included in Figure 10 for reference. [28] is much slower than the execution on XLFDD or DCPMM. We believe it is because of the large amount of read/write to flash memory, and furthermore, it does not use low latency flash, and access to edge data is not fast. Note that, [28] is complementary to XLFDD, and there is a possibility that further speedup can be achieved by combining the two. The main focus of [28] is sequentialization of vertex updates so that it is suitable for writing in flash memory, while XLFDD can provide fast random read accesses, for example, to CSR format.

6 DISCUSSIONS

Measures to improve IOPS. While we have achieved 99 MIOPS with nine XLFDDs in this paper, some applications may require more IOPS as exemplified by BFS on kron30 shown in Figure 6. A simple way to increase IOPS is to use more XLFDDs: for example, 24 XLFDDs can be connected via PCIe Switch in a single server. Another possibility is to increase the number of dies on XLFDD. These measures do not improve *IOPS per capacity*. That is, they may not be suitable for cases where very large capacity is not required, in which case we can take an approach of operating multiple planes simultaneously [45]. This is feasible with a small increase in die cost and is also applicable to low latency flash memory such as XL-FLASH. Therefore, there is a possibility that a flash memory drive with several to several tens of times higher IOPS can be introduced in the near future without significantly increasing capacity or cost.

Switching top-down/bottom-up. For some graph processing algorithms, optimizations have been proposed that switch the processing strategy when the active vertex set is sparse and when it is dense [17, 23, 47]. This also applies to the switching between top-down BFS and bottom-up BFS mentioned in Section 5.1. We focus on speeding up the processing of random accesses with XLFDD, so we fix it to top-down BFS, but there is a possibility that performance will be improved even when using flash memory by switching top-down and bottom-up. In bottom-up processing, sequential access to `edge_data` may be effective. As described in Section 1, sequential access can be performed fast enough using flash memory. Therefore, it is possible to optimize access pattern to the flash memory according to top-down phase and bottom-up phase.

Workloads with sequential and/or write access. In this paper, we show some cases where performance close to that of DRAM can be achieved even with microsecond-level random read accesses for flash memories. However, for a practical system, it is necessary to deal with sequential and write access as well as random read. For sequential reads, existing interfaces such as NVMe provide sufficient performance. On the other hand, for workloads with write

access, such high performance is difficult to achieve. There are two drawbacks for performance. First, the write latency of flash memory is much longer than that of read. Second, because write must be done in block unit of several MBytes in flash memory, small-sized write results in read-modify-write. In addition to the reduced overhead by the proposed method, workloads with write accesses may require changes to make them more suitable for flash memory. Since sequential write or bulk write are suitable for flash memory, it is possible that a combination of sequential/bulk writes and random reads may provide better performance. We leave these studies for future work.

Logical-to-physical conversion. XLFDD is a prototype and does not have logical-to-physical address conversion. Therefore, physical addressing is necessary and the implementation efforts described in Section 4.3.1 are effective to reduce CPU overhead. However, these are not necessary if XLFDDs are equipped with logical-to-physical conversion on the drive, like normal SSDs. Even in this case, we believe that application performance is not much different from our experimental results. This is because even if there is a hardware-level conversion, there will be little difference in CPU overhead from our results, and while the conversion may increase read latency a bit, it can be small enough compared to microsecond latency.

Advantages of low latency flash memory over SCM. This paper introduced a method for approaching in-memory execution times with memories having latencies of up to microseconds. In terms of application performance, either SCM or low latency flash memory is fine to use, but flash memory has the advantage of being able to handle a larger capacity than SCM. While the currently available SCM, Optane DCPMM, has a maximum capacity of 4.5 TB per socket [6], PCIe connected devices using flash memory can expand to tens of TB. With the proposed method, we believe that low-latency flash memory can be a high-capacity alternative to SCM.

7 RELATED WORK

Emerging memories and hiding access latency. There is a great deal of research on memory with performance between DRAM and conventional flash memory. These include PCM, MRAM and low latency flash memory, for example. Among them, Optane DCPMM/SSD series [7] and XL-FLASH [15] are the commercialized ones with large capacity. These have a read latency of several hundreds of nanoseconds to several microseconds. Several methods have been proposed to perform efficient access while hiding the latency. [30] shows an implementation of KVS to achieve near in-memory performance with user-level context switches and low latency NVMe SSDs. However, its target is several MIOPS, which is an order of magnitude less than our work. In [21], the access by DMA and the access by load or prefetch instruction are compared, which concludes that accesses by load instructions are promising because DMA requires a heavy queue operation. However, the current CPU has an upper limit on the number of in-flight memory accesses issued by the load instruction, resulting in saturated performance for microsecond-latency devices. In order to hide the microsecond-level latency, [40] proposes a new processor architecture that can switch multiple contexts at the hardware level. In terms of hiding access latency, [26] proposes to use stackless coroutine to hide the latency in DRAM case. Also, [29] utilizes coroutines to hide the DRAM latency for in-memory graph processing.

Graph Processing. If the graph fits in the DRAM capacity, it can be processed in-memory like Galois [41], GAP [18], and so on [23, 47]. However, if the graph does not fit in the DRAM capacity, it must be placed on disks, or placed on DRAMs of multiple machines for distributed processing [38]. However, it is reported that the performance of distributed processing deteriorates due to the communication overhead [22, 39]. For this reason, a number of methods have been proposed in which graph data is placed on disks such as SSDs and processed by a single machine (Graphene [35], flashgraph [48], Mosaic [37], BigSparse [27] and GraFBoost [28], X-stream [43], GraphChi [31]). Graphene achieves the excellent performance close to that of in-memory using SSDs in workloads with many sequential accesses (e.g. PageRank). However, it is significantly slower than in-memory for workloads with many small random accesses such as BFS [35]. In [32], it is shown that the performance can be improved even in a workload such as BFS by preprocessing the graph so that the edge data cached in DRAM is easy to hit. However, significant improvements in the performance of disk-based BFS-like algorithms would require a large DRAM. There are also attempts to process large-scale graphs while suppressing performance degradation from DRAM by using SCM [36, 44] or Optane DCPMM [24, 42].

8 CONCLUSION

In this paper, we show that microsecond-level latency flash memory can be used to achieve performance close to that of in-memory in some read-intensive and small-sized random access workloads. Instead of existing NVMe interface, we propose a new lightweight interface to reduce CPU overhead. A read request can be processed with a CPU overhead of 6.7 ns. In addition, to hide the latency of flash memory, we operate hundreds of contexts per CPU core by using stackless coroutines. The cost of context switches among the stackless coroutines are negligible. Combining the new interface and the coroutines enables the performance 99 MIOPS per CPU core, which is demonstrated on PCIe-connected prototype boards, XLFDD, with the microsecond-latency flash. We apply the proposed method to large-scale graph processing in which small-sized random accesses frequently occur. We use GAP benchmark suite which is one of the existing in-memory graph processing frameworks as a baseline. Note that the DRAM replacement by XLFDD can be applied to other in-memory graph processing implementations as well. BFS, BC and SSSP are executed with XLFDD. The execution time with XLFDD is about 88% to 141% of in-memory. Our proposal has an advantage over DRAM for BFS-like graph processing with required IOPS in the range of tens of MIOPS and low access locality. This will be the first example of how flash memory can be used to achieve the same or better performance as in-memory in applications that involve a large number of small-sized random read accesses. We believe that the proposed method can be applied to other applications that require many small-sized random read accesses to large memory spaces such as databases.

REFERENCES

[1] [n.d.]. Web Data Commons - Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph/. visited on 2021/3/23.

[2] [n.d.]. Compute Express Link (CXL). https://www.computeexpresslink.org/. visited on 2021/3/23.

[3] [n.d.]. Graph 500. large-scale benchmarks. https://graph500.org/. visited on 2021/3/23.

[4] [n.d.]. Intel Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html. visited on 2021/3/23.

[5] [n.d.]. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. visited on 2021/3/23.

[6] [n.d.]. Intel Optane Persistent Memory 200 Series Brief. https://www.intel.co.jp/content/dam/www/public/us/en/documents/product-briefs/optane-persistent-memory-200-series-brief.pdf. visited on 2021/3/26.

[7] [n.d.]. Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html. visited on 2021/3/23.

[8] [n.d.]. NVM Express (NVMe). https://nvmexpress.org/. visited on 2021/3/23.

[9] [n.d.]. Storage Performance Development Kit. https://spdk.io/. visited on 2021/3/23.

[10] [n.d.]. WebGraph Dataset. http://webgraph.di.unimi.it/. visited on 2021/3/23.

[11] 2017. Intel Optane Memory Series Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/99742/intel-optane-memory-series-32gb-m-2-80mm-pcie-3-0-20nm-3d-xpoint.html. visited on 2021/3/23.

[12] 2017. Ultra-Low Latency with Samsung Z-NAND SSD. https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf. visited on 2021/3/23.

[13] 2018. Intel Optane SSD 800P Series Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/125298/intel-optane-ssd-800p-series-118gb-m-2-80mm-pcie-3-0-3d-xpoint.html. visited on 2021/3/23.

[14] 2019. 10.39M Storage I/O Per Second From One Thread. https://spdk.io/news/2019/05/06/nvme/. visited on 2021/3/23.

[15] 2019. Toshiba Memory Corporation Introduces XL-FLASH Storage Class Memory Solution. https://about.kioxia.com/en-jp/news/2019/20190806-1.html. visited on 2021/3/23.

[16] 2020. KIOXIA CM6-R Series. https://business.kioxia.com/en-jp/ssd/enterprise-ssd/cm6-r.html. visited on 2021/3/23.

[17] Scott Beamer, Krste Asanovic, and David A. Patterson. 2012. Direction-optimizing breadth-first search. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, Jeffrey K. Hollingsworth (Ed.). IEEE/ACM, 12.

[18] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619

[19] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[20] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[21] Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman, and Nima Honarmand. 2018. Taming the Killer Microsecond. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 627–640.

[22] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 752–768.

[23] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, Christian Scheideler and Jeremy T. Fineman (Eds.). ACM, 393–404.

[24] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2019. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *CoRR* abs/1904.07162 (2019). arXiv:1904.07162

[25] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714

[26] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin J. Levandoski, and Gor V. Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.* 11, 11 (2018), 1702–1714.

[27] Sang Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2017. BigSparse: High-performance external graph analytics. *CoRR* abs/1710.07736 (2017). arXiv:1710.07736

[28] Sang Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi (Eds.). IEEE Computer Society, 411–424.

[29] Vladimir Kiriansky, Yunming Zhang, and Saman P. Amarasinghe. 2016. Optimizing Indirect Memory References with milk. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*, Ayal Zaks, Bilha Mendelson, Lawrence Rauchwerger, and Wen-mei W. Hwu (Eds.). ACM, 299–312.

[30] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 1–15.

[31] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 31–46.

[32] Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam H. Noh, and Jiwon Seo. 2019. Pre-Select Static Caching and Neighborhood Ordering for BFS-like Algorithms on Disk-based Graph Engines. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 459–474.

[33] Jure Leskovec, Deepayan Chakrabarti, Jon M. Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2008. Kronecker Graphs: An Approach to Modeling Networks. *CoRR* abs/0812.4905 (2008). arXiv:0812.4905

[34] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Proceedings of the Workshop on Experimental Computer Science, Part of ACM FCRC, San Diego, CA, USA, 13-14 June 2007*. ACM, 2.

[35] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 285–300.

[36] W. Liu, H. Liu, X. Liao, H. Jin, and Y. Zhang. 2019. NGraph: Parallel Graph Processing in Hybrid Memory Systems. *IEEE Access* 7 (2019), 103517–103529.

[37] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 527–543.

[38] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 135–146.

[39] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland.

[40] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. 2019. Enhancing Server Efficiency in the Face of Killer Microseconds. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 185–198.

[41] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 456–471.

[42] Ivy Bo Peng, Maya B. Gokhale, and Eric W. Green. 2019. System evaluation of the Intel optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2019, Washington, DC, USA, September 30 - October 03, 2019*. ACM, 304–315.

[43] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 472–488.

[44] M. Shantharam, K. Iwabuchi, P. Cicotti, L. Carrington, M. Gokhale, and R. Pearce. 2017. Performance Evaluation of Scale-Free Graph Algorithms in Low Latency Non-volatile Memory. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1021–1028.

[45] N. Shibata, K. Kanda, T. Shimizu, J. Nakai, O. Nagao, N. Kobayashi, M. Miakashi, Y. Nagadomi, T. Nakano, T. Kawabe, T. Shibuya, M. Sako, K. Yanagidaira, T. Hashimoto, H. Date, M. Sato, T. Nakagawa, H. Takamoto, J. Musha, T. Minamoto, M. Uda, D. Nakamura, K. Sakurai, T. Yamashita, J. Zhou, R. Tachibana, T. Takagiwa, T. Sugimoto, M. Ogawa, Y. Ochi, K. Kawaguchi, M. Kojima, T. Ogawa, T. Hashiguchi, R. Fukuda, M. Masuda, K. Kawakami, T. Someya, Y. Kajitani, Y. Matsumoto, N. Morozumi, J. Sato, N. Raghunathan, Y. L. Koh, S. Chen, J. Lee, H. Nasu, H. Sugawara, K. Hosono, T. Hisada, T. Kaneko, and H. Nakamura. 2019. 13.1 A 1.33Tb 4-bit/Cell 3D-Flash Memory on a 96-Word-Line-Layer Technology. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*. 210–212.

[46] T. Shiozawa, H. Kajihara, T. Endo, and K. Hiwada. 2020. Emerging Usage and Evaluation of Low Latency FLASH. In *2020 IEEE International Memory Workshop (IMW)*. 1–4.
[47] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, 135–146.
[48] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 45–58.