NVM-Optimized Graph
CSC2224 Final Report

KyoKeun Park(1002294812) and Siyue Wang(1006720988)

December 10, 2020

Abstract

Adoption of Intel Optane DC Persistent Memory has remained relatively slow, even though it promises interesting features. We believe part of it has to do with the lack of direct compatibility with existing programs. The alternative would be to modify an existing program and modify it to utilize the Optane memory. In this paper, we optimize an existing graph processing system, Ligra, and show that we can sustain its excellent performance, with minimal amount of modification. We show that the modification has added approximately 1,000 lines of code, and such modification allows Ligra to process huge graphs which does not fit in memory, while performing better than some of well-known out-of-core graph processors.

1 Introduction

The non-volatile memory (NVM) promises DRAM-like performance, with a cheaper cost and persistency. Intel has been at the front-line with their Intel Optane DC Persistent Memory [6], when it comes to pushing NVM to the industry. As such, there has been numerous number of work in recent years in an attempt to utilize NVM with existing data structures [5, 8, 19, 20]. It has been shown through these work that existing data structures can be modified to perform efficiently with NVM with some NVM-specific optimizations. Some optimizations aim to improve the reliability of data structures in NVM [5], while others aim directly to improve its performance with the help of NVM [8, 19, 20]. Although there has been many valiant effort in optimizing structures, such as B-tree and hashmaps, there has not been many work towards optimizing graph processing systems for NVM.

Although Intel Optane persistent memory promises interesting performance metric while providing high capacity for relatively low price, it appears that its adoption has been quite slow. We believe its slow adoption is related to code refactoring that must be done from application developer side in order for it to fully leverage the persistent memory. In this paper, we will demonstrate our implementation of graph processing system, which also leverages Intel Optane persistent memory. We have done so by taking an existing graph processor and modifying it. As such, we also show that minimal code refactoring can be done from application developer's side to leverage persistent memory by maintaining the graph processor's interface.

2 Background

In this section, we discuss the design of Intel Optane persistent memory and the performance evaluation, which has been conducted by previous papers. Furthermore, we will discuss different types of graph processing system and what could potentially be done in order for it to utilize the persistent memory.

2.1 Intel Optane DC Persistent Memory

Note that any reference to NVDIMM, NVM, or persistent memory in the paper would be a reference to Intel Optane DC Persistent Memory. Persistent memory can be configured in multiple modes:

Memory Mode uses persistent memory as large main memory and use DRAM as a large cache. In this mode, Optane loses its persistency. Memory mode tends to be the simplest mode for application developers however, since no additional work is required from developer's side for their application to work with it. Of course, using persistent memory as a volatile memory would mean that it loses the benefits of persistence. Hence, we will not be focusing on memory mode for this paper.

App-Direct Mode exposes the persistent memory directly to the Operating System. Applications can utilize it through Intel's provided PMDK library [14]. Within the app-direct mode, there are two different configurations: **FsDAX** and **DAX** mode. With FsDAX, persistent memory acts as a regular block device. As such, applications must go through some kind of filesystem in order to access the persistent memory. Do note that some filesystems with direct-access support (ie., Ext4 [18] and XFS [3]), can create a special byte-addressable file. In DAX mode, entire persistent memory is displayed as a byte-addressable device to the Operating System, which differs from both regular memory and block device. Applications can utilize the entire persistent memory as byte-addressable storage rather than part of it, without worrying about the overhead of filesystem code path.

From previous literature on performance evaluation of persistent memory [7], we know that access latency and bandwidth is noticeable slower when compared to DRAM, but order of magnitude faster when compared to SSDs. Furthermore, it has been discovered that sequential accesses can perform as much as 2x when compared to random accesses, and its write performance is much worse than read performance. Hence, the performance characteristics of Intel Optane indicates that the ideal type of data to store in it would be read-only data that is read sequentially. Lastly, one of the interesting characteristics of the persistent memory is that they can be accessed in cacheline granularity, which is 64-bytes. However, it has been found that accessing at 256-byte granularity will lead to higher performance.

2.2 Graph Processing Systems

Different graph processing systems tend to have different approach, which also leads to different memory access patterns. However, they can essentially be divided into two different types: vertex-centric and edge-centric.

Vertex-centric graph processors iteratively performs computation over vertices. For each vertex, the value of local vertex gets propagated to its neighbours. Vertex-centric graph processors include Ligra [16] and GraphChi [9].

Edge-centric graph processors iteratively performs compuation over edges, rather than vertices. This approach is commonly used in order to avoid random access to edges. Edge-centric graph processors include X-Stream [15] and GridGraph [21].

Another type of graph processor, which is not mutually exclusive to previous two, is **out-of-core** graph processor. They are disk-based single-machine graph processing systems, and are seen as a cheaper alternative to a distributed graph processing [15]. By utilizing the large storage devices, some out-of-core graph processors has proven to be quite performant, even when compared to some of the large distributed graph processing units [21]. X-Stream and GridGraph again falls into this category.

3 Motivation

As discussed briefly in Section 1, the adoption of Intel Optane persistent memory has been quite slow since its release. In order to assist its adoption, we hope to show that with minimal modification, that existing applications and/or interfaces could utilize the NVM.

Moreover, we noticed that there has been minimal work revolving around graph processors with NVM in mind, specifically Intel Optane persistent memory. We have also observed that graph processors tend to write the graph data to memory once (preprocessing) and read from it for the rest. Furthermore, graph processors tend to store graphs in sequential manner. Hence, we believed that we could leverage the characteristics of persistent memory heavily with graph processors.

4 Previous Work

Previous result of not persisting all the fields in common data structures like doubly linked list into has been shown to make a significant performance difference when we are using persistent memory.[12]

Graphmat [17] is a shared memory graph processing framework. It uses sparse vectors, vertex associated data and matrices to represent its graph. It has been shown that Graphmat is able to achieve close to DRAM performance by simply putting vertices vectors into main memory and putting other data structures such as matrices into NVM. However, the experiment uses NUMA-aware software HEMP that can make NVM and DRAM appear as two memory nodes [13].

HyVE [4], which aims to be a graph processing system focused on being used with ReRAM [1], another type of NVM, which is said to be different than Intel Optane DC Memory. We were not able to confirm this due to the fact that Intel refuses to provide any details regards to its architecture. Although both HyVE and we aim to design a graph data structure, different types of NVMs are targeted. Furthermore, HyVE puts heavy emphasis on designing its graph system to be energy-efficient.

There are many works [2, 4] have been done on B+ Trees that are designed for persistent memory because B+ Trees play an important role in databases and file systems. However, there has not been much done for general graph processing frameworks. Even though previous results stated above have demonstrated that the graph representation and the memory allocation can cause a difference in a hybrid system that have both persistent memory and DRAM, no graph framework has been found to utilize direct access mode of persistent memory to represent the graph for better performance.

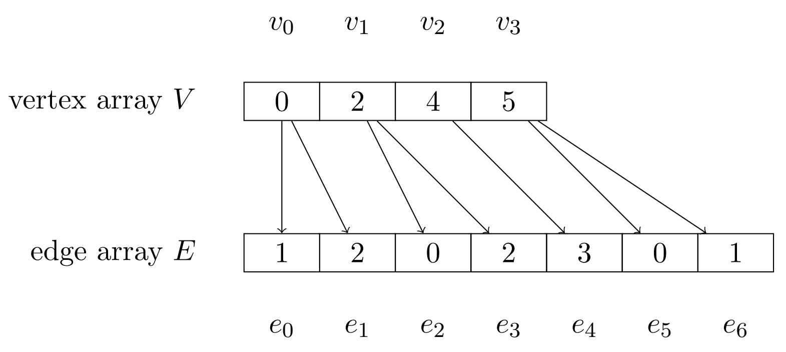
5 Design

We extended the existing framework **Ligra**[16] to support persistent memory. Ligra is a shared memory vertex-centric graph processing framework with simple and clear interfaces, which made it suitable as our base framework. Ligra uses adjacency list as its graph representation, and the whole graph is in DRAM. Thus, in order to achieve the best performance possible under a hybrid system that has both persistent memory and DRAM, a change to the graph representation is necessary due to the above reasons. We change the structure of vertex in the original Ligra, and we add an array to represent all the edges for the graph. Since the edges are accessed in a small, sequential chunks, and they tend to be written once, we believe that it is ideal to store them into the persistent memory.

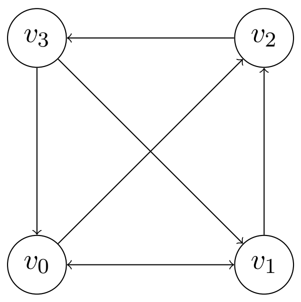
Suppose we have a graph $G(V, E)$ with n vertices and m edges. We reference $V[i]$ as v_i, and we call the index of v_i its ID. Each vertex v_i has an offset o_i, and v_i has $o_{i+1} - o_i$ outedges. If $i = n-1$, then v_i has $m - o_i$ outedges. v_i's neighbor IDs on the other end of outedges are stored in edge array E starting from offset o_i. Figure 1 shows an example. Figure 1(a) shows a simplified version of how the graph representation is like, and Figure 1(b) shows a visualization of the graph. In Figure 1(a), vertex array is only showing each vertex's offset as the value. $o_0 = 1$ and $o_1 = 2$, so v_0 has two outedges. And because v_0's offset is 0, its outedges start from index 0 in edge array E. As you can see, $E[0] = 1$ and $E[1] = 2$, so we have $(0, 1)$ and $[0, 2]$ outedges from v_0.

6 Implementation

Our modification to Ligra has taken 1k LoC, where majority of the code was simply a port from existing code in order to implement objects which were used by applications to interact with Ligra. Hence, our modification to Ligra has been quite minimal. In order to access NVM, we used Intel's PMDK library [14]. Since Ligra interface has remain unchanged, applications which uses Ligra framework also required minimal modification for it to work



(a) Example graph representation



(b) Example graph

Figure 1: New design of graph representation

with NVM. The notable changes that had to be made is that the path of Optane memory and its size now need to be specified, and NVM-specific objects must be used.

7 Experiments

For our experiments, we focus on the performance and the scalability of our implementation. We compare our implementation with GridGraph[21] and the original Ligra [16]. GridGraph is an out-of-core graph processing framework. It preprocesses the input file and partitions the input file into grids. Their experiments show that they outperform other state of art out-of-core graph processing frameworks, X-stream [15] and GraphChi [9].

7.1 Configuration

We have conducted experiments on a 16-core machine with 512 Gb Optane persistent memory and 64 DRAM. The persistent memory is configures as FsDAX mode, meaning we use special direct access enabled files for experiments.

We decide to benchmark four algorithms, BFS(Breadth first Search), PageRank, MIS(Maximal independent set) and Radii(finding the biggest radius of the graph). We have done experiments on two kinds of large graphs, real world networks from the Stanford University [10] and randomly generated graphs from Ligra's random graph generator.

We sample at least 10 executions and their execution time for each benchmark and use the average run time for comparison. All executions of BFS starts with the same node and PageRank is set to 100 iterations.

We also chose the optimal partition number for GridGraph, since the number of partitions can influence how well GridGraph performs. We also conduct experiments by simply putting the preprocessing files of GridGraph into the persistent memory. We reference to it as GridGraph on NVM in the experiment results.

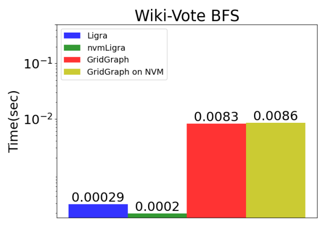
7.2 Results

Results are shown from Figure 2 to Figure 6. In case of it is hard to read the legend, the blue bar is ligra, the green bar is nvmLigra, our implementation of ligra, the red bar is GridGraph and the yellow bar is putting GridGraph's preprocessing files in the persistent memory. Note that the y-axis is on log scale.

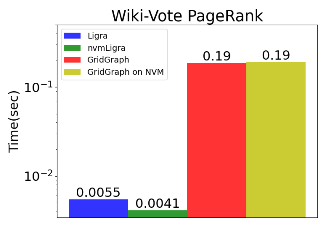
7.2.1 Real World Networks

Our experiments use wiki-Vote, wiki-Talk and twitter networks from the Stanford University dataset. We want to investigate that if our performance might change, based on the network's size and density.

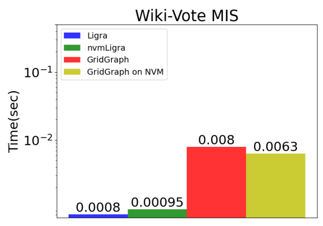
Wiki-Vote has 7115 vertices and 103,689 edges. It is the smallest network we experiment on. Experiment results on Wiki-Vote is shown in Figure 2.



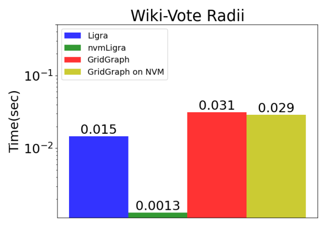
(a) Wiki-Vote BFS



(b) Wiki-Vote PageRank

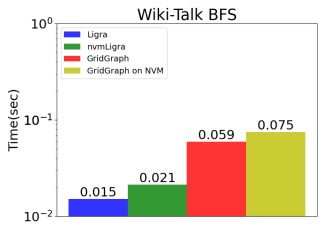


(c) Wiki-Vote MIS

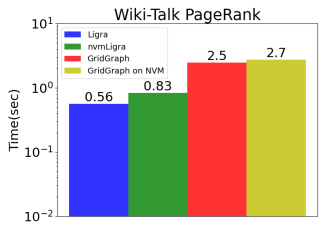


(d) Wiki-Vote Radii

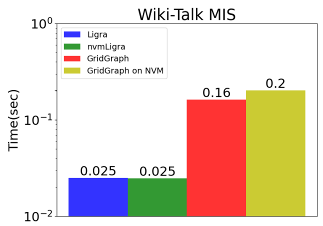
Figure 2: Wiki-Vote



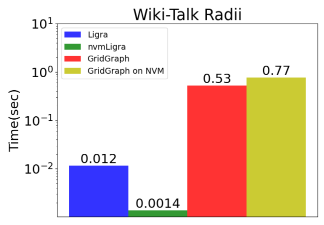
(a) Wiki-Talk BFS



(b) Wiki-Talk PageRank



(c) Wiki-Talk MIS



(d) Wiki-Talk Radii

Figure 3: Wiki-Talk

Surprisingly, nvmLigra outperforms in BFS, PageRank and Radii. However, when we look back to experiment result dataset, some Ligra execution takes unusually long time, which makes the average see, worse than our nvmLigra.

Wiki-Talk has 2,394,385 vertices and 1,768,149 edges. It a much more sparse network with more vertices and edges than Wiki-Vote. Experiment results on Wiki-Vote is shown in Figure 4. Overall, nvmLigra performs very similar to the original Ligra.

Twitter has 81,306 vertices and 5,021,410 edges. It is denser than both Wiki-Vote and Wiki-Talk. Experiment results on twitter is shown in Figure 4. As you can see the results, GridGraph outperforms us on BFS, but it failed to execute PageRank. Ligra and nvmLigra shares similar performance again on BFS, MIS and Radii.

In conclusion, Ligra outperforms nvmLigra overall on real world networks if we are not going to consider outliers, and nvmLigra shares very similar performance to Ligra after all.

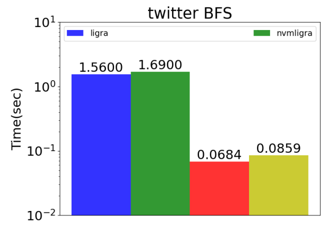
7.2.2 Randomly Generated Networks

Due to the DRAM limitation of the machine, we were not able to find larger real networks to do comparison against Ligra and GridGraph. Therefore, we randomly generated two networks, rMat_100,000,000 which has 100 million edges and rMat_billion which has a billion random edges.

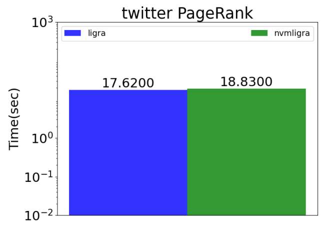
Results are shown in Figure 5 and Figure 6. In experiments on this two experiments, we can clearly see that Ligra outperforms nvmLigra except rMat_100,000,000 by very little.

7.2.3 Preprocessing files in NVM & Performance

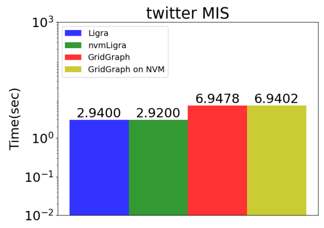
We also found an interesting phenomenon from the experiments above that putting GridGraph's preprocessing files in the persistent memory will make the performance even worse, and as you can see in Figure 6, Radii has worsened 5 times. We have con-



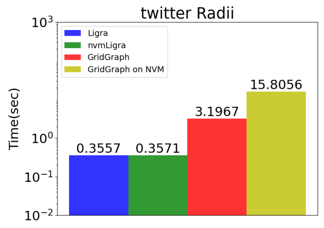
(a) twitter BFS



(b) twitter PageRank

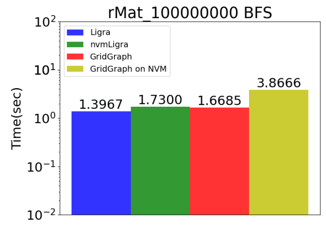


(c) twitter MIS

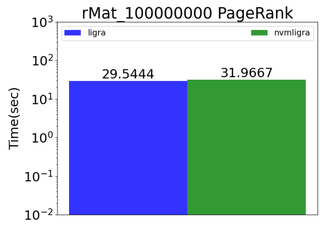


(d) twitter Radii

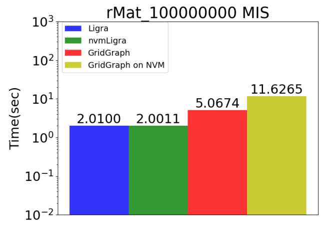
Figure 4: twitter



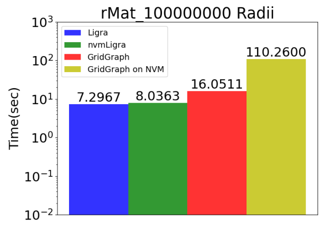
(a) rMat_100,000,000 BFS



(b) rMat_100,000,000 PageRank



(c) rMat_100,000,000 MIS



(d) rMat_100,000,000 Radii

Figure 5: rMat_100,000,000

ducted experiments on GraphChi which is also an out-of-core graph processing framework, and it preprocesses its input by partitioning them into shards. Since GraphChi does not have BFS, MIS, or Radii available, we have only conducted experiments on PageRank. We refer GraphChi/GridGraph on NVM as putting their preprocessing filse in the persistent memory as well, while GraphChi/GridGraph on SSD means putting their prepricessing files on SSD.

Table 1 shows that the running time in seconds for GridGraph and GraphChi . We can see GraphChi on NVM performs better in Wiki-Vote, while it performs worse than GraphChi on SSD.

We also suspect that the number of partitions might play a role in the performance downgrade. In Table 2, we can see that the number of partitions do not seem to play a role for at least GraphChi. GraphChi 10 shards on SSD and GraphChi 10 shards on NVM do not seem to make a huge difference except wiki-Vote.

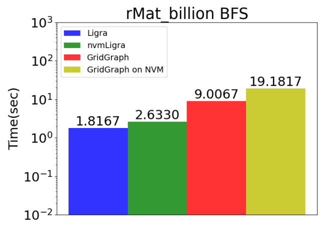
Further investigations are needed for out-of-core frameworks like GraphChi and GridGraph. However, due to the time limit of this project and the lack of root access, we are not able to perform more experiments regarding this phenomenon.

7.2.4 1.8billion Edges Large Graph

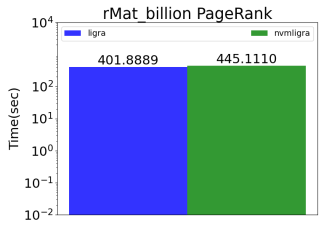
We also experimented on a network, com-friendster, from the Stanford University Dataset. com-friendster has 1,806,067,135 edges and 65,608,366 vertices. Ligra is not able to fit the whole input into DRAM, and the system kills the process because it is trying to use more memory than the system has. However, our nvmLigra is able to process the network since we use the persistent memory to fit all of our edges.

7.3 Summary

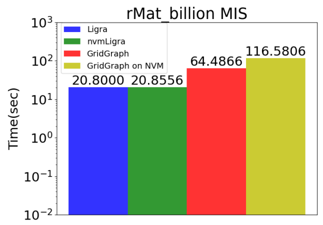
We can see the conclusion from experiments even though sometimes our implementation of Ligra outperforms the original Ligra, it is mostly because of some outliers when running the original Ligra. Over-



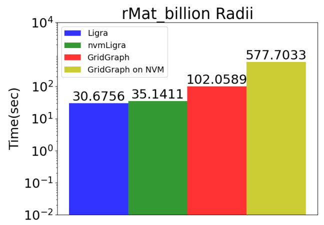
(a) rMat_billion BFS



(b) rMat_billion PageRank



(c) rMat_billion MIS



(d) rMat_billion Radii

Figure 6: rMat_billion

GridGraph & GraphChi				
	GridGraph on SSD	GridGraph on SSD on NVM	GraphChi	Graphchi on NVM
wiki-Talk(100 iterations)	2.4678	2.7444	36.62184	37.58734444
wiki-Vote(100 iterations)	0.1867	0.1911	7.1983	6.5277

Table 1: Results of running GridGraph[21] and GraphChi[9] on SSD vs Persistent Memory

all, the original Ligra performs better, which is expected since DRAM's write and read latency are a lot better than persistent memory. But we can see that The utilization of direct access of persistent memory does not hurt performance that much even though its read latency is 3 times slower than DRAM for random access and 2 times slower than DRAM for sequential access. At the same time, it is cheaper than DRAM.

8 Limitations and Future Work

There still exist some limitations in our work. For example, when the graph can fit in the main memory, it is for sure DRAM will outperform the persistent memory since DRAM's read and write latency are much smaller. Some future work can also be done.

8.1 DRAM Caching

Inspired by X-stream[15] and Mosaic[11], prefetching some edge information from the persistent memory to DRAM can be done. A streaming interface such as having an asynchronous thread to work like a buffer can be implemented. We did not have enough time to finish this during the span of this course.

8.2 Preprocessor

A preprocessor can be implemented since once we processed the graph, a file containing all the edges is created in the persistent memory. The preprocessor can simply read from that file in the persistent memory the next time we are going to analyze the same input file. This way, we reap the benefit of persistence, as well as its performance.

8.3 Experiment with Larger Graphs

Due to lack of time and trouble with running larger graphs on our system, we were not able to conduct a full experiment with them. As such, more experiments with larger graphs can be conducted, since our implementation can fit in the input that is larger than the size of DRAM, but smaller than the size of the persistent memory.

GraphChi on SSD & GraphChi on nvm regarding partitions				
	GraphChi	GraphChi on NVM	GraphChi 10 shards	GridGraph on NVM 10 shards
wiki-Talk(100 iterations)	36.62184	37.5873	89.12724444	89.9151
wiki-Vote(100 iterations)	7.1982	6.527652222	62.8212	67.4515
twitter(4 iterations)	139.2189	143.1892	139.7966	139.6217

Table 2: Results of running GraphChi[9] with 1 shard and 10 shards on SSD vs Persistent Memory

9 Conclusion

Throughout the course, we have explored the characteristics of Intel Optane memory and discussed potentially different ways of utilizing it for graph processing systems. We have initially started our project with creating entirely new graph processor in mind. But then after the first literature review, we realized that it may be a better idea to build on top of existing graph processor instead in order to demonstrate the minimal work required to adopt this new technology.

In the end, we have shown that it is possible to optimize existing graph processors to be compatible with Optane persistent memory. We have also shown that it performs closer to the performance of in-memory graph processors compared to out-of-core processors, while supporting larger graphs then what in-memory graph processors could handle. We have also shown that we can achieve this respectable performance without heavy optimization within Ligra and almost no change to the application itself. If similar approach can be taken for other data structures, we believe it could encourage the adoption of NVM.

10 Acknowledgement

We would like to thank Prof. Gennady Pekhimenko and Hossein Farrokhbakht for guiding us on our project throughout the course. Furthermore, we would like to thank one of Prof. Pekhimenko's student, Han Jie Qiu, for not only lending us the system with necessary hardware to test our project on a bare metal, but also providing us with necessary information related to NVM to get us started. Without them, we would not have been able to conduct the experiments shown in this paper.

References

[1] Hiroyuki Akinaga and Hisashi Shima. "Resistive random access memory (ReRAM) based on metal oxides". In: *Proceedings of the IEEE* 98.12 (2010), pp. 2237–2251.

[2] Ping Chi, Wang-Chien Lee, and Yuan Xie. "Adapting B+-Tree for Emerging Nov-volatile Memory Based Main Memory". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35 (Jan. 2015), pp. 1–1. DOI: 10.1109/TCAD.2015.2512899.

[3] Dave Chinner. *xfs: DAX Support*. Mar. 2015. URL: https://lwn.net/Articles/637715/.

[4] Tianhao Huang et al. "HyVE: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing". In: *2018 Design, Automa-*

tion & Test in Europe Conference & Exhibition (DATE). IEEE. 2018, pp. 973–978.

[5] Deukyeon Hwang et al. “Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree”. In: *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 187–200. ISBN: 978-1-931971-42-3. URL: `https://www.usenix.org/conference/fast18/presentation/hwang`.

[6] *Intel and Micron Produce Breakthrough Memory Technology*. 2015. URL: `https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/`.

[7] Joseph Izraelevitz et al. “Basic Performance Measurements of the Intel Optane DC Persistent Memory Module”. In: (Mar. 2019).

[8] Wook-Hee Kim et al. “ClfB-Tree: Cacheline Friendly Persistent B-Tree for NVRAM”. In: *ACM Trans. Storage* 14.1 (Feb. 2018). ISSN: 1553-3077. DOI: `10.1145/3129263`. URL: `https://doi.org/10.1145/3129263`.

[9] 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. Hollywood, CA, USA: USENIX Association, 2012, pp. 31–46. ISBN: 9781931971966.

[10] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014.

[11] Steffen Maass et al. “Mosaic: Processing a trillion-edge graph on a single machine”. In: *Proceedings of the Twelfth European Conference on Computer Systems*. 2017, pp. 527–543.

[12] Pratyush Mahapatra, Mark D. Hill, and Michael M. Swift. *Don’t Persist All : Efficient Persistent Data Structures*. 2019. arXiv: `1905.13011 [cs.DB]`.

[13] Jasmina Malicevic et al. “Exploiting NVM in Large-Scale Graph Analytics”. In: *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*. INFLOW ’15. Monterey, California: Association for Computing Machinery, 2015. ISBN: 9781450339452. DOI: `10.1145/2819001.2819005`. URL: `https://doi.org/10.1145/2819001.2819005`.

[14] *Persistent Memory Programming*. URL: `https://pmem.io/`.

[15] 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*. 2013, pp. 472–488.

[16] Julian Shun and Guy E Blelloch. “Ligra: a lightweight graph processing framework for

shared memory". In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 135–146.

[17] Narayanan Sundaram et al. "GraphMat: High Performance Graph Analytics Made Productive". In: *Proc. VLDB Endow.* 8.11 (July 2015), pp. 1214–1225. ISSN: 2150-8097. DOI: `10.14778/2809974.2809983`. URL: `https://doi.org/10.14778/2809974.2809983`.

[18] Matthew Wilcox. *DAX: Page cache bypass for filesystems on memory storage*. Oct. 2014. URL: `https://lwn.net/Articles/618064/`.

[19] Xingbo Wu et al. "NVMcached: An NVM-based Key-Value Cache". In: *APSys '16*. 2016.

[20] Jun Yang et al. "NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems". In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 167–181. ISBN: 978-1-931971-201. URL: `https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang`.

[21] 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)*. Santa Clara, CA: USENIX Association, July 2015, pp. 375–386. ISBN: 978-1-931971-225. URL: `https://www.usenix.org/conference/atc15/technical-session/presentation/zhu`.