Co-design of Advanced Architectures for Graph Analytics using Machine Learning

Kuldeep Kurte, Neena Imam, Ramakrishnan Kannan, S.M.Shamimul Hasan, Srikanth Yoginath

Computing and Computational Sciences Directorate
Oak Ridge National Laboratory, Oak Ridge, TN, USA
Email: {kurtekr, imamn, kannanr, hasans, yoginathsb}@ornl.gov

***Abstract*—A graph is an excellent way of representing relationships among entities. We can use graph analytics to synthesize and analyze such relational data, and extract relevant features that are useful for various tasks such as machine learning. Considering the crucial role of graph analytics in various domains, it is important and timely to investigate the right hardware configurations that can achieve optimal performance for graph workloads on future high-performance computing systems. Design space exploration studies facilitate the selection of appropriate configurations (e.g. memory) to achieve a desired system performance. Recently, the approach of accelerating graph analytics using persistent non-volatile memory has gained a lot of attention. Traditional system simulators such as Gem5 and NVMain can be used to explore the design space of these advanced memory architectures for graph workloads. However, these simulators are slow in execution thus limiting the efficiency of design space exploration studies. To overcome this challenge, we proposed a machine learning based approach to co-design advanced memory architectures for graph workloads. We tested our approach with DRAM, non-volatile memory, and hybrid memory (DRAM+NVM) using a breadth first search benchmark algorithm. Our results showed the applicability of the proposed machine learning based approach to the co-design of the advanced memory architectures. In this paper, we provide recommendations on selecting advanced memory architectures to achieve desired performance for graph workloads. We also discuss the performances of different machine learning models that were considered in this study.**

***Index Terms*—Design Space Exploration; Graph Analytics; Non Volatile Memory; Machine Learning;**

I. Introduction

In this paper we present a co-design methodology for computer micro-architectures for the optimum performance of graph analytics workloads. Our methodology is based on Machine Learning (ML) and is applied for the design of computer memory architectures to optimize the performance of graph analytics workloads. However, our methodology can be used for applications other than graph analytics. For computer architecture design, it is necessary to choose appropriate configurations to satisfy various performance, power, temperature, reliability, and other metrics. This process is known as the Design Space Exploration (DSE). The size of the micro-architecture design space has been growing at a rapid pace due to the increasing complexity of the modern computer systems and the complex interactions among various hardware components, software stack, and application softwares. Traditionally, computer architects utilize large-scale cycle-accurate architectural simulators that are used on representative benchmarks to explore the design space. Some examples of such architectural simulators used by computer vendors include the Mambo simulation environment [1], the SimNow simulator [2], and the HAsim simulator [3]. However, the speed of these traditional architectural simulators is extremely slow. To improve the simulator's speed and fidelity, several enhanced simulators have been developed [4]. Nevertheless, even the enhanced simulators cannot meet the demands of simulating and optimizing the performance of extreme scale computing systems. Various approaches have been proposed to overcome these limitations, including several ones based on machine learning for more efficient design space exploration. We utilized machine learning methods for micro-architectural design to build a predictive model by utilizing a small set of simulated configurations in the training phase. Such predictive models in essence approximate the simulator function that characterizes the relationship between the design parameters and processor responses. Some example design parameters for memory architectures are cache size and queue size. Example processor response parameters are performance and energy consumption. Then, in the predicting phase, the trained models are used to predict the responses of new design configurations that are not involved in the training set. Since simulations are only required in the training phase, the machine learning techniques are relatively efficient than other traditional approaches when they can accurately predict the micro-architectural performance while employing a small labeled training set.

We developed an end-to-end workflow for the co-design of computer memory architectures and graph analytics workloads. Our workflow combines traditional simulators such as the Gem5 [5] and the NVMain simulators [6] with ML to make recommendations for optimized hardware configurations

for graph analytics workloads. In Section II we present some technical background and related work. Section III describes our end-to-end workflow. Section IV presents detailed results and discussion of our experimental results. We conclude by discussing our planned future research.

II. Background and Related Work

The purpose of this research is to develop an end-to-end workflow for optimum performance of graph analytics applications to be executed on computing platforms with hierarchical memory structures. Historically, the main memory component of computer architectures has been the ubiquitous Dynamic Random-Access Memory (DRAM). DRAMs are present in mobile hand-held devices as well as in the most powerful supercomputers. However, conventional computer memory technologies will not be able to meet the challenges of future extreme-scale systems. These challenges include energy efficiency, system reliability, and application performance. There is an increasing gap between the frequencies of CPUs and the latencies of memory systems. This gap, known as the memory wall [7], results in performance bottleneck. Additionally, DRAM device scaling has also plateaued to result in lower on-node memory capacity. The limited on-node memory capacity forces the applications to engage in increased inter-node communications resulting in degradation of application performance and system reliability. In addition, the current memory systems are not energy efficient due to the fact that they consume power even in idle mode. Traditional DRAM main memory systems consume as much as 30-50%, of the total power budget of a computing system [8]. Therefore, alternative memory technologies such as various non-volatile memory concepts [9] are being investigated. NVM devices such as the NAND flash memories are deployed on state-of-the-art supercomputers. U.S. Department of Energy's Summit Supercomputer (currently ranked 2 on the TOP500 list [10]) has 1600 GB of NVM in each of its 4608 nodes and these can be configured either as burst buffers or as extended memory [11]. However, the optimum integration of NVMs in a computer memory architecture is an active and open area of research. NVMs can be integrated in multiple ways in memory hierarchy. NVMs can be added to augment on-node DRAM, as global storage devices, as CPU cache, or as I/O burst buffers. Therefore, rigorous design space exploration of these emerging memory systems is necessary such that appropriate choices of memory parameters/configurations can be made to satisfy performance, power, reliability, and other metrics.

The objectives of our paper are optimum hardware configuration selection and design space exploration for graph analytics applications. Researchers have investigated potential acceleration of graph analytics using persistent/non-volatile memory technologies (e.g., NVM SSD and byte-addressable NVM) and proposed Metall and miniVite tools [12], [13]. The miniVite and Metall tools showed significant performance improvements on NERSC Cori and OLCF Summit supercomputers by employing a graph that was persistently stored instead of regenerated [12]. The NVG_{RAPH} data structure was proposed to support in-memory graph storage and computing for non-volatile main memory systems [14]. Experimental results showed NVG_{RAPH}'s performance to be comparable to the Compressed Sparse Row Representation (CSR) and Linked-Node Analytics using Large Multiversioned Arrays (LLAMA) in-memory data structures. High-performance graph analytics performance on Intel's Optane DC Persistent Memory (Optane PMM) was studied in [15]. The results showed that the Optane PMM yields competitive performance on large production clusters by supporting a range of efficient algorithms. Dhulipala proposed a semi-asymmetric graph engine called Sage and showed that shared memory and non-volatile memory implementations of graph algorithms can solve a wide range of graph problems [16]. Kumar and Huang proposed SafeNVM, which offered a reliable NVM store to support application-specific data formats such as databases and persistent key-value stores [17]. A parallel graph processing framework for a hybrid memory system (DRAM + NVM) called NGraph was proposed in [18]. The authors reported that NGraph was 48.28% faster than a lightweight graph processing framework for shared memory called Ligra [19] and a numa-aware graph-structured analytics platform called Polymer [20] systems. These prior promising results certainly motivate computer scientists to further explore the optimum memory configurations (DRAM, NVM, and DRAM+NVM) for large scale graph analytics workloads.

Traditional architectural-level simulator-based memory design approach is often inefficient due to the significant computational costs. The problem further intensifies for emerging architectures with complex interactions among hardware and application software. To overcome these challenges, ML-based methods have been applied to design various micro-architectural aspects, such as the memory controller optimization [21] and memory hierarchy [22]. The ML-based DSE approaches can be significantly more efficient than traditional simulator-based DSE methods.

Therefore, in this work, we propose and evaluate a ML-based DSE approach for graph analytics workloads. We applied ML approach to build predictive models that approximate the functionality of an architectural simulator by providing the relationship between the memory configuration parameters (e.g., number of channels, ranks, banks, controller frequency, timing parameters) and memory responses (e.g., latency, bandwidth, power, number of read/write operations). Our previously published work concentrated on ML-based DSE of hybrid memory design for applications such as the HPCG and STREAM benchmarks [23]. This paper builds upon our previous work to design an end-to-end ML-based DSE workflow specifically for graph analytics benchmarks such as the Graph500 [24]. Graph analytics applications have different memory access patterns compared to traditional HPC simulation workloads. As such, the workflow presented in this paper provides valuable insights for the optimization of future extreme scale systems for graph analytics.

III. ML-BASED MEMORY CO-DESIGN FOR GRAPH WORKLOADS

In this section, we present our end-to-end workflow for co-design of advanced memory architectures for graph analytics applications. We describe various components of this workflow and discuss their respective roles. Further, we present various machine learning algorithms considered for this study followed by a description of the overall experimental setup.

A. Co-design workflow for advanced memory architectures for graph workloads

We implemented an end-to-end semi-automated memory co-design workflow for conducting machine learning based hybrid memory co-design exploration of graph data related workloads. The pictorial view of our memory co-design workflow for graph workload is shown in Figure 1. The workflow consists of three major components: 1) Gem5 system simulator, 2) non-volatile memory simulator called NVmain, and 3) machine learning models.

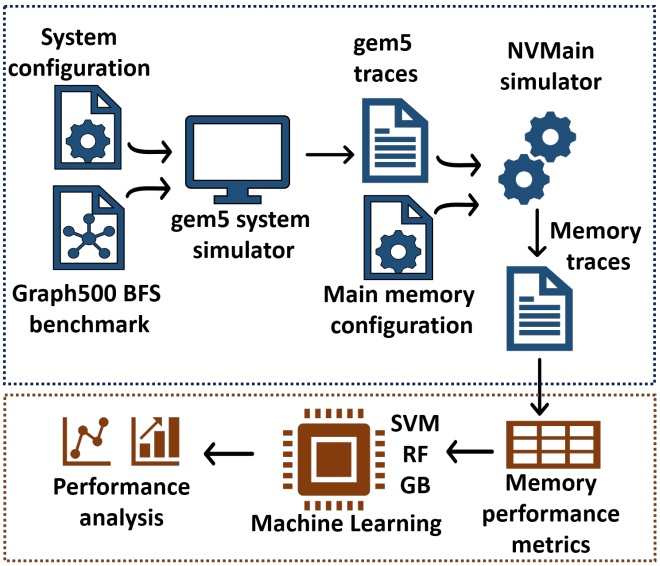


Fig. 1: End-to-end workflow for co-design of advanced memory architectures for graph workloads using ML

- **Gem5 system simulator**: Our objective is to simulate the memory responses using NVMain simulator. To achieve this, we need memory traces of any workload executing in a computer system. Gem5 is a computer-system simulator which can be used to simulate a designated workload as a sequence of discrete events [5]. These events consist of both compute events as well as memory access events. The Gem5 simulator outputs these events in the form of traces (both compute and memory event traces). The Gem5 simulator can be used in two modes: 1) System Emulation (SE) mode which simulates the system calls and services and 2) Full System (FS) mode in which a complete system with all devices and operating system can be simulated. We specify to the Gem5 simulator the system configuration (i.e. CPUs, memory size, etc.) via a system configuration file and provide the executable of the application workload (e.g. a benchmark algorithm). Subsequently, Gem5 simulates the benchmark and produces the trace file. We further extract the memory-related traces and convert them to the NVMain compatible format.
- **NVMain simulator**: NVMain is an architectural-level simulator capable of simulating various memory performance parameters such as energy, bandwidth, latency, etc. with cycle-accurate operations of main memory designs using both DRAM and emerging NVM technologies including their hybrid designs [6]. Depending on the simulation type, i.e. DRAM, NVM, or hybrid, we specify the configuration parameters to the NVMain simulator. These configuration parameters broadly consist of memory architectural parameters such as number of channels in the memory module, number of ranks per channel, number of memory banks, number of rows and columns in each bank, clock frequency of the memory interconnect, and CPU frequency. Similarly, the configuration parameters also include timing related parameters such as column read time (*tBURST*), data restoration time (*tRAS*), row activation time (*tRCD*), pre-charge time (*tRP*), write to pre-charge time (*tWR*), etc. Along with these configuration parameters, we can also specify a few control parameters that are related to the NVMain's operation such as PrintGraphs, TraceReader, PrintPreTrace, etc.

We provide the memory configuration file and memory trace file (extracted from Gem5 trace) to NVMain to simulate the memory performances. NVMain's output consists of various performance metrics such as memory bandwidth, total data reads and writes, total power and energy, etc. The following are the definitions of few of these metrics which we considered in this work. Typically, the memory performance depends on the memory configuration parameters as well as the type of workload (in our case graph processing algorithm).

- **Latency**: It is the time between a processor initiating a read (or write) request for a byte or word and receiving (or successfully writing into the memory) the byte or word. The lower is the latency of a memory the better is the performance. We can extract total as well as average values for the latency from NVMain's output trace.
- **Bandwidth**: It is defined as a rate at which the data can be read from and written into the memory. For example, the number of bytes read (or written) per second.
- **Power**: This indicates the total power in Watts consumed by the memory unit while executing the specified benchmark.
- **Memory read and write operations**: These values represent the total number of read and write operations performed while executing the specified workload. Endurance of a memory is related to the write operation. While DRAM is considered to have infinite endurance (on the order of 10^{15}), NVMs have finite endurance of

the order of 10^8–10^9 [6].

Next, we combine the memory performance parameters with the corresponding memory configuration parameters to generate the data set for training machine learning models. Afterwards, the pre-trained ML models can be used to predict the memory performance for graph benchmarks for various memory configuration parameters.

B. Machine learning algorithms considered in this work

The specific objective of co-designing the advanced memory architectures for graph workload using machine learning is to understand the relationships between the memory configuration parameters and memory performance metrics using ML approach. We develop the surrogate ML models that can be used for the co-design of advanced memory architectures for graph workloads. The main limitations of using system and memory simulators such as Gem5 and NVMain are their time expensive nature of execution. One potential approach that gained momentum in recent years is to develop surrogate ML models. Once such models are available, they can be used to quickly predict the performance of a particular memory configuration for a given benchmark. This approach has a potential to accelerate the co-design process. In this work we have used the following ML algorithms and regression techniques for the co-design of advanced memory architectures.

- **Support Vector Machines** (SVM): SVM provides a set of supervised algorithms for classification, regression, and outlier detection [25]. We are using SVM as a regressor. SVM is a linear non-probabilistic binary classifier which can be applied to the non-linearly separable data through the kernel trick and can be easily applied to the multi-class classification problem. In principle, SVM tries to assign each training data point to one of the two classes so that the difference between the two classes is maximized.
- **Random Forest** (RF): The next ML algorithm considered is random forest [26]. It is based on an ensemble learning method that creates a large number of decision trees such that the parameters of each of these trees are randomly perturbed. Each tree is overfitted and later the results of all the trees are combined to obtain the final result. We are using the *RandomForestRegressor* function provided in the scikit-learn library.
- **Gradient Boosting** (GB): This algorithm is based on a principle of fitting several weak learners (such as simple decision trees) iteratively with the modified training data in each iteration [27]. Initially, all the data points are assigned equal weights and the learners are trained. In the subsequent iterations, the weights of the data points that were predicted less accurately in the previous iteration are increased compared to the data points that were predicted correctly. As a result, the learners are forced to learn to predict the data points which are hard to learn. We use the *GradientBoostingRegressor* function provided in the scikit-learn library.

C. Overall setup for graph workloads

Our objective is to co-design the advanced memory architectures for graph workloads using machine learning. Here, we describe the graph algorithm and the overall end-to-end setup of our workflow.

The pictorial view of our co-design workflow for graph benchmark can be seen in Figure 1. We computed the Breadth-First Search (BFS) kernel as specified in the Graph500 benchmark by starting from a random vertex ID. The Graph500 is a large-scale benchmark for HPC platforms. Instead of a computation-intensive benchmark like the High Performance Linpack (HPL) [28], the Graph500 is focused on data-intensive workloads [24]. We used a synthetic graph generator called the GTGraph [29] to generate a graph with 1,024 vertices and with an edge factor of 16. Next we ran our BFS code on the generated data in the Gem5 simulator with the default system configuration in a System Emulation mode (SE). This run produced a trace file with the size of 5GB. The next task is to extract the traces related to the memory operations and simulate them in the NVMain simulator.

Further, we used NVMain to simulate the performance of main memory using DRAM, NVM, and hybrid memory configurations [6]. Each NVMain simulation for a particular memory configuration took around 2 hours. In order to avoid human errors, we automated the process of generating configuration files for 1) pure DRAM, 2) pure NVM, and 3) a hybrid (combinations of DRAM and NVM) with different numbers of channels as well as with different values for various memory configuration related parameters. We specifically considered CPU frequency, memory controller frequency, fraction of memory, and two timing parameters, i.e. *tRAS* and *tRCD*. We used CPU frequencies of 2 GHz, 3 GHz, 5 GHz, and 6.5 GHz. We also used controller frequencies of 400 MHz, 666 MHz, 1250 MHz, and 1600 MHz. Using our configuration generation scripts, we generated several configuration files for three types of memory configurations.

Next, we created a comprehensive dataset from NVMain's output trace files for training ML models. We created a post-processing script to extract the memory performance values from the NVMain's output trace. The memory performance parameters we considered are latency, bandwidth, memory reads and writes, and power. We combined these values with their corresponding memory configuration parameters to create a comprehensive dataset which we further used for ML model training.

D. Challenges faced for the co-design workflow for graph workload

We faced the following challenges in executing the proposed workflow:

- The Graph500 benchmark like Scott Beamers Graph Algorithm Platform (GAP) [30], did not run on the Gem5 simulator. Some of the advanced C++ runtime capabilities are not supported in the Gem5 simulator while operating in SE mode. We observed that some system calls such as *mprotect*, *set_robust_list*, *rt_sigaction*, *rt_sigprocmask*,

and *sched_getaffinity* were unavailable in Gem5, which produced warnings and exceptions. Moreover, in the Gem5 simulator, the graph was wrongly produced, with 537 vertices and 0 edges in place of 1,024 vertices with 10,468 edges. Furthermore, the BFS program did not run to completion. To overcome this challenge we computed the BFS kernel as specified in the Graph500 benchmark by starting from a random vertex ID. We used a synthetic graph generator called the GTGraph to generate a graph with 1,024 vertices and with an edge factor of 16.

- Typically, a sequential approach processes the Gem5 trace file one line at a time in a sequence. We observed that our Gem5 trace file for a graph algorithm has over $\approx$ 91.5M lines. The sequential processing of this file is a time consuming task. To overcome this challenge, we employed a parallel script to convert the Gem5 traces into the memory trace format compatible with NVMain. A multiprocessing module in *Python* was used for parallel computing. This parallel script divides the input file into chunks. The size of each chunk is user specified and the starting points of these chunks are provided to the parallel processes. Each parallel process processes its own chunk. Finally, each process stores its memory trace lines sequentially in a list and this list is further stored in a file. Our parallel script showed a linear speed-up over the sequential approach to extracting the memory-related traces from Gem5 output. The procedure created a 14GB-sized output file in a required format for the NVMain simulator.

IV. Results and Discussion

In this section we present the experimental results. First, we describe the experimental setup used for ML based co-design for advanced memory architectures for graph workload followed by memory performance metrics and statistics used for the performance evaluation. Next, we discuss various observations that can be drawn from the results where we also provide recommendations for the co-design of advanced memory architectures.

A. Experimental setup

1) Objectives of the experiments: The objective is to understand the relationships between the memory architectural parameters and memory performance metrics and learn about the suitability of ML algorithms for co-designing advanced memory architectures for graph algorithms in three memory modes, i.e. DRAM, NVM, and hybrid (DRAM+NVM).

2) Simulation setup: We used Gem5 system simulator in SE mode to simulate the graph benchmark and extracted memory traces from Gem5 output. For this work, we used Gem5's default configuration in SE mode which uses atomic CPU and atomic memory access. Next, we ran the NVMain simulator to simulate the memory traces. We used four different CPU frequencies 2 GHz, 3 GHz, 5 GHz, and 6.5 GHz and four different controller frequencies 400 MHz, 666 MHz, 1250 MHz, and 1600 MHz. We considered 2 and 4 channels to generate memory configurations for NVMain simulations. Additionally, we used data restoration time (*tRAS*) and row activation time (*tRCD*) as timing parameters. For DRAM, we used *tRAS* value of 24 cycles and *tRCD* value of 9 cycles. The *tRAS* for NVM was 0 as non-volatile memories do not need to restore the data similar to DRAMs. We used several *tRCD* values for NVM depending on the controller frequency. For 400 MHz of controller frequency we used *tRCD* values of 20, 30, 40, 50, 60, and 80 cycles. For 666 MHz of controller frequency we used *tRCD* values from 33, 50, 67, 83, 100, and 133 cycles. For 1250 MHz of controller frequency we used *tRCD* values from 62, 94, 125, 156, 187, and 250 cycles. Lastly, for 1600 MHz of controller frequency we used *tRCD* values of 80, 120, 160, 200, 240, and 320 cycles.

3) Data: In this work we computed the BFS kernel as specified in the Graph500 benchmark by starting from a random vertex ID. We selected this algorithm because it is data intensive in nature and it is one the most representative algorithms in graph analytics workloads. We used a synthetic graph generator called GTGraph to generate a graph with 1,024 vertices and with an edge factor of 16. For this work, we limited our analysis to BFS algorithms with graph of 1024 vertices. However, in future, we will experiment with different graph algorithms and various sizes of the input graph. We generated a comprehensive dataset for ML model training from outputs of NVMain simulations. Out of total 416 memory configurations, for a few configurations NVMain simulation exited with segmentation fault error. At this moment we don't know the reason for this segmentation fault. We selected around 374 data points, i.e. the memory configurations which ran successfully, and used 80% of these data points for training the ML models, and 20% data points for testing. Figure 2 summarizes the dataset used for the ML model training. The first three columns of the table represent memory configuration parameters. For brevity we are showing only three configurations parameters CPU frequency *CPUFreq*, controller frequency (*ControlFreq*), and number of channels (*nCh*). The columns 4–9 represent average values of six memory performance metrics for each memory type, i.e. DRAM (*D*), NVM (*N*), and hybrid (*H*). The color coding depicts darker shades for higher values and lighter shades for lower values of performance metrics. This representation is helpful for comparing the performance of different memory types having different memory configurations. We provide our detailed observations in the subsection IV-B.

4) ML algorithms and performance metrics: We mainly used three machine learning algorithms 1) Support Vector Machine (SVM) 2) Random forest (RF), and 3) Gradient boosting trees (GB) as regressors. In this setting, the memory configuration parameters represent the predictor variables and memory performance parameters represent the predicted variables. The training set is composed of these predictor variables and their corresponding predicted variables. Further, the ML algorithms use mean square error (MSE) as a loss function (i.e. error function) which is a convex function. The task of ML algorithms is to minimize this loss function. We evaluated

CPU Freq	Control Freq	nCh	Average Power			Average Bandwidth			Average Latency			Average Total Latency			Average Memory Reads			Average Memory Writes		
			D	N	H	D	N	H	D	N	H	D	N	H	D	N	H	D	N	H
2000	400	2	0.17	0.04	0.11	985.12	877.46	958.92	31.87	26.58	27.39	168.29	874.48	544.16	4.13E+07	4.13E+07	4.13E+07	4.48E+06	4.48E+06	4.48E+06
2000	400	4	0.15	0.04	0.10	530.79	477.80	484.74	29.81	23.72	24.72	125.61	1265.01	2070.16	2.06E+07	2.06E+07	2.06E+07	2.24E+06	2.24E+06	2.24E+06
2000	666	2	0.16	0.06	0.09	1107.87	983.39	906.71	31.87	28.18	22.98	168.29	1227.83	605.97	4.13E+07	4.13E+07	3.44E+07	4.48E+06	4.48E+06	3.74E+06
2000	666	4	0.15	0.06	0.04	582.26	535.71	184.71	29.81	24.46	8.29	125.61	1754.45	1274.39	2.06E+07	2.06E+07	6.88E+06	2.24E+06	2.24E+06	7.47E+05
2000	1250	2	0.15	0.12	0.07	1214.16	1072.06	587.49	31.87	31.87	14.25	168.30	2011.20	1292.82	4.13E+07	4.13E+07	2.06E+07	4.48E+06	4.48E+06	2.24E+06
2000	1250	4	0.14	0.11	0.11	624.83	584.48	494.59	29.81	26.19	21.49	125.61	2837.92	6564.72	2.06E+07	2.06E+07	1.72E+07	2.24E+06	2.24E+06	1.87E+06
2000	1600	2	0.15	0.15	0.15	1243.94	1095.67	1212.17	31.87	34.16	28.56	168.29	2484.79	3136.67	4.13E+07	4.13E+07	4.13E+07	4.48E+06	4.48E+06	4.48E+06
2000	1600	4	0.14	0.15	0.10	636.44	597.65	406.71	29.81	27.26	17.30	125.61	3491.61	9237.47	2.06E+07	2.06E+07	1.38E+07	2.24E+06	2.24E+06	1.49E+06
3000	400	2	0.18	0.04	0.12	1297.68	1117.62	1254.33	31.87	26.58	27.39	168.29	734.74	441.99	4.13E+07	4.13E+07	4.13E+07	4.48E+06	4.48E+06	4.48E+06
3000	400	4	0.16	0.04	0.07	716.85	623.50	425.72	29.81	23.72	16.41	125.61	1039.58	699.57	2.06E+07	2.06E+07	1.38E+07	2.24E+06	2.24E+06	1.49E+06
3000	666	2	0.17	0.06	0.12	1519.45	1295.62	1479.18	31.87	28.18	27.61	168.29	995.18	578.90	4.13E+07	4.13E+07	4.13E+07	4.48E+06	4.48E+06	4.48E+06
3000	666	4	0.15	0.06	0.05	814.04	725.92	374.70	29.81	24.46	12.48	125.62	1379.12	750.48	2.06E+07	2.06E+07	1.03E+07	2.24E+06	2.24E+06	1.12E+06
3000	1250	2	0.15	0.12	0.12	1726.78	1454.49	1379.21	31.87	31.87	23.90	168.29	1574.53	1134.68	4.13E+07	4.13E+07	3.44E+07	4.48E+06	4.48E+06	3.74E+06
3000	1250	4	0.14	0.11	0.11	899.74	818.51	704.26	29.81	26.19	21.21	125.61	2133.47	4288.62	2.06E+07	2.06E+07	1.72E+07	2.24E+06	2.24E+06	1.87E+06
3000	1600	2	0.15	0.15	0.10	1787.63	1498.46	1149.37	31.87	34.16	19.22	168.29	1925.87	1329.27	4.13E+07	4.13E+07	2.75E+07	4.48E+06	4.48E+06	2.99E+06
3000	1600	4	0.14	0.15	0.15	924.01	844.60	873.02	29.81	27.26	26.13	125.59	2589.91	6427.66	2.06E+07	2.06E+07	2.06E+07	2.24E+06	2.24E+06	2.24E+06
5000	400	2	0.20	0.04	0.08	1739.10	1431.34	1093.19	31.87	26.58	18.20	168.29	622.96	287.16	4.13E+07	4.13E+07	2.75E+07	4.48E+06	4.48E+06	2.99E+06
5000	400	4	0.17	0.04	0.09	996.23	824.72	704.95	29.81	23.72	20.58	125.60	859.24	653.84	2.06E+07	2.06E+07	1.72E+07	2.24E+06	2.24E+06	1.87E+06
5000	666	2	0.18	0.06	0.11	2162.01	1737.80	1723.13	31.87	28.18	23.12	168.29	809.06	400.29	4.13E+07	4.13E+07	3.44E+07	4.48E+06	4.48E+06	3.74E+06
5000	666	4	0.16	0.06	0.09	1194.38	1014.01	886.98	29.81	24.46	20.88	125.61	1078.85	1519.11	2.06E+07	2.06E+07	1.72E+07	2.24E+06	2.24E+06	1.87E+06
5000	1250	2	0.16	0.12	0.14	2607.48	2037.53	2501.34	31.87	31.87	28.87	168.29	1225.20	681.93	4.13E+07	4.13E+07	4.13E+07	4.48E+06	4.48E+06	4.48E+06
5000	1250	4	0.15	0.12	0.09	1388.44	1204.53	848.43	29.81	26.19	16.99	125.61	1569.90	1594.38	2.06E+07	2.06E+07	1.38E+07	2.24E+06	2.24E+06	1.49E+06
5000	1600	2	0.16	0.15	0.13	2748.76	2125.49	2187.22	31.87	34.16	24.68	168.29	1478.73	661.15	4.13E+07	4.13E+07	3.44E+07	4.48E+06	4.48E+06	3.74E+06
5000	1600	4	0.15	0.15	0.15	1447.08	1261.98	1348.06	29.81	27.26	25.94	125.61	1868.55	3495.13	2.06E+07	2.06E+07	2.06E+07	2.24E+06	2.24E+06	2.24E+06
6500	400	2	0.20	0.04	0.07	1971.21	1585.58	927.64	31.87	26.58	13.80	168.29	584.26	189.01	4.13E+07	4.13E+07	2.06E+07	4.48E+06	4.48E+06	2.24E+06
6500	400	4	0.17	0.04	0.07	1151.58	928.46	633.37	29.81	23.72	16.42	125.60	796.81	573.42	2.06E+07	2.06E+07	1.38E+07	2.24E+06	2.24E+06	1.49E+06
6500	666	2	0.19	0.07	0.11	2532.76	1971.14	2009.29	31.87	28.18	23.18	168.29	744.63	353.85	4.13E+07	4.13E+07	3.44E+07	4.48E+06	4.48E+06	3.74E+06
6500	666	4	0.16	0.06	0.12	1424.88	1175.55	1230.46	29.81	24.46	24.94	125.61	974.91	1262.57	2.06E+07	2.06E+07	2.06E+07	2.24E+06	2.24E+06	2.24E+06
6500	1250	2	0.17	0.12	0.10	3166.50	2367.16	2014.30	31.87	31.87	19.10	168.29	1104.28	386.57	4.13E+07	4.13E+07	2.75E+07	4.48E+06	4.48E+06	2.99E+06
6500	1250	4	0.15	0.12	0.09	1709.93	1439.70	1032.17	29.81	26.19	17.07	125.61	1374.82	907.23	2.06E+07	2.06E+07	1.38E+07	2.24E+06	2.24E+06	1.49E+06
6500	1600	2	0.16	0.15	0.08	3377.31	2487.25	1581.52	31.87	34.16	15.01	168.29	1323.95	340.97	4.13E+07	4.13E+07	2.06E+07	4.48E+06	4.48E+06	2.24E+06
6500	1600	4	0.15	0.15	0.10	1799.74	1522.64	1098.07	29.81	27.26	17.44	125.62	1618.85	1063.61	2.06E+07	2.06E+07	1.38E+07	2.24E+06	2.24E+06	1.49E+06

Fig. 2: Summary of memory performance metrics. First three columns represent memory configuration parameters. The columns 4–9 represent average values of memory performance metrics where 'D', 'N', and 'H' represent DRAM, NVM, and hybrid memories respectively. Color coding depicts darker shades for high values and lighter shades for low values.

the suitability of these algorithms in predicting the memory performances for a given memory configuration. We used mean squared error (*MSE*) (Eq. 1) and R^2-score (coefficient of determination) (Eq. 2) statistics to compare the trained ML model's performance on test data points. In Eq. 1 and 2, y and $\hat{y}$ represent simulated values and predicted values (by ML models) of memory performance metrics respectively. Also, $\bar{y}$ represents the mean of the simulated memory performance metric and n is the total number of datapoints considered for testing. ML model is considered better if it shows a smaller value for *MSE* and R^2 value closer to 1.0.

$$MSE(y, \hat{y}) = \frac{1}{n}\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1}$$

$$R^2(y, \hat{y}) = 1 - \frac{\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2}{\Sigma_{i=1}^{n}(y_i - \bar{y}_i)^2} \tag{2}$$

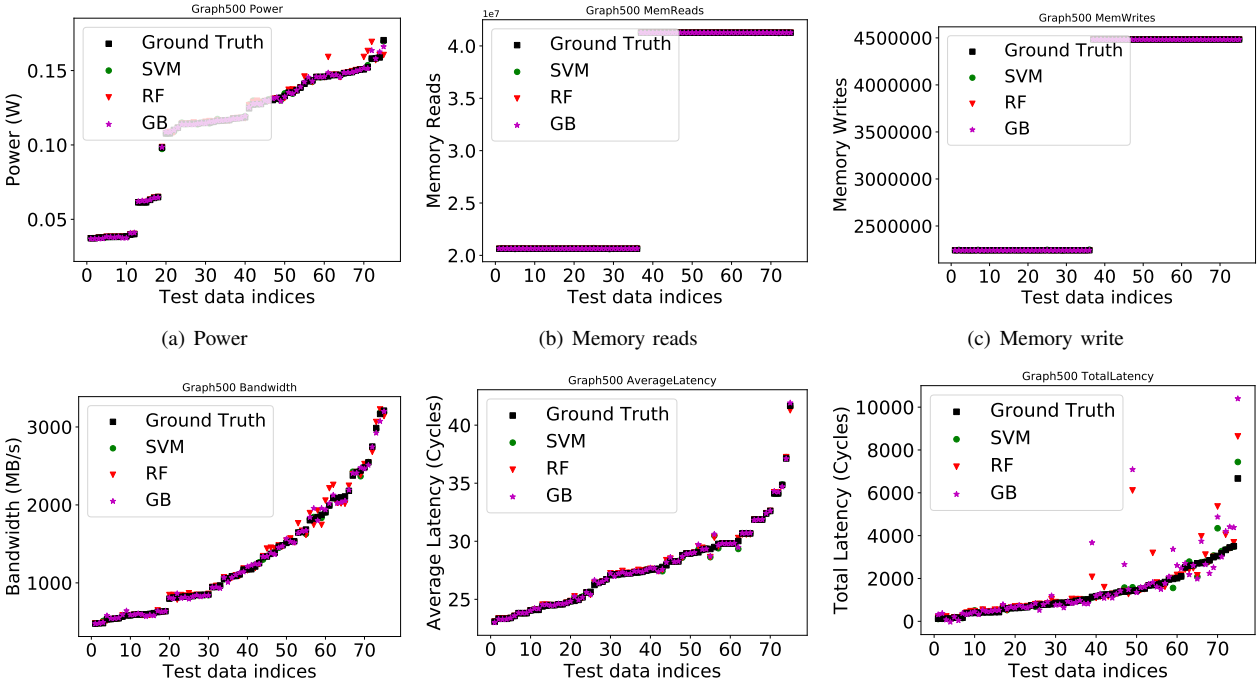
B. Results and observations

In this section we discuss and summarize the memory performance metrics that are collected from the outputs of NVMain simulations for graph workload (shown in Figure 2). We also pinpoint various patterns in memory performances based on CPU frequency, controller frequency, and number of channels. Further, we compare the performance of ML algorithms detailed in Section III-B for predicting memory performance metrics of bandwidth, power, average latency, total latency, memory reads, and memory writes for the Graph500 Breadth-First Search (BFS) benchmark. Figure 3 shows six plots, each corresponding to the scatter plot of the respective performance metric for three ML algorithms along with the ground truth. Table I lists MSE and R^2 statistics for different ML models for predicting memory performance metrics. Table I also highlights the best performing algorithm for a given memory performance metric.

For a fair comparison, it is required that the values of different performance metrics are normalized to the same scale. For example, power takes values between $[0, 1]$ whereas the values of memory reads and writes are in the range of 10^7 (refer Figure 2). There are different normalization techniques such as a) Z-normalization that uses mean μ and standard deviation σ of observations and b) min-max normalization that transforms the minimum value in observations to 0 and the maximum value to 1. In this paper, we used a min-max scalar technique to scale values of performance metrics in the ML training dataset.

From the summary of memory performance metrics shown in Figure 2 along with ML model performances shown in Figure 3 and the performance metric Table I, we gain following insights into the advanced memory architectures.

1) Power: NVMain outputs total power for each channel. We calculated the average power per channel for the purpose of ML training. We reported in Figure 2 the mean values of



(a) Power (b) Memory reads (c) Memory write

(d) Bandwidth (e) Average latency (f) Total latency

Fig. 3: Graphs comparing the performance of ML models in predicting the values of different memory performance metrics with ground truth for various memory configurations in the test data. From left-top corner to bottom-right corner: (a) Bandwidth (b) Memory reads (c) Memory writes (d) Power (e) Average latency, and (f) Total latency. The ground truth data was obtained by simulating the memory configurations in the test data. The x-axis of the plots represent the indices of the test data. The y-axis represents the corresponding memory performance metric. Overall, each graph plots the value of a specific memory performance metric by three ML models (SVM, RF, and GB) along with the ground truth for each memory configuration in the test data.

average power per channel for various memory configurations described by the first three columns. Overall, we observed that DRAM consumed more power, NVM consumed less power and hybrid memory consumed average power for all memory configurations. This was expected as DRAM requires power for operations such as pre-charging and data restoration which are absent in NVMs. DRAM showed a slightly decreased trend of average power consumption per channel for higher controller frequencies for a given CPU frequency while NVM showed a reverse trend. Hybrid memory showed a mixed trend similar to DRAM until 666MHz of controller frequency and showed a reverse trend for 1250MHz and 1600MHz controller frequencies. Finally, for our graph benchmark, NVMs with controller frequency of 400 MHz showed better performance for average power consumption per channel for a given CPU frequency and its performance was independent of number of channels.

TABLE I: ML model's performance on graph benchmark

Memory Parameters	Statistics	Linear	SVM	RF	GB
Bandwidth	MSE	5.59×10^{-3}	3.95×10^{-5}	3.80×10^{-4}	1.42×10^{-4}
	R2	9.07×10^{-1}	9.99×10^{-1}	9.94×10^{-1}	9.98×10^{-1}
Memory Reads	MSE	1.10×10^{-13}	5.82×10^{-7}	7.00×10^{-12}	9.87×10^{-8}
	R2	1.00	1.00	1.00	1.00
Memory Writes	MSE	9.14×10^{-12}	6.75×10^{-7}	5.94×10^{-10}	$1.03\times10^{-}7$
	R2	1.00	1.00	1.00	1.00
Power	MSE	7.28×10^{-3}	1.91×10^{-5}	3.10×10^{-4}	4.95×10^{-5}
	R2	8.69×10^{-1}	1.00	9.94×10^{-1}	9.99×10^{-1}
Average Latency	MSE	2.05×10^{-3}	1.04×10^{-4}	1.01×10^{-4}	1.32×10^{-4}
	R2	9.41×10^{-1}	9.97×10^{-1}	9.97×10^{-1}	9.96×10^{-1}
Total Latency	MSE	2.86×10^{-3}	1.56×10^{-4}	1.91×10^{-3}	3.35×10^{-3}
	R2	3.86×10^{-1}	9.66×10^{-1}	5.90×10^{-1}	2.80×10^{-1}

In Figure 3(a), we can see that SVM and GB fit the test data well for average power consumption per channel. Also, we can verify from Table I that SVM's MSE is the lowest with $R^2 = 1$. RF was unable to characterize the differences between hybrid, NVM and DRAM in slightly higher power ranges, such as power values closer to 0.15W. These observations lead to the conclusion that SVM is a better choice for characterizing the average power consumption per channel in co-designing the advanced memory architectures for graph workloads.

2) Memory reads/writes: NVMain typically provides memory read and write values for each channel, i.e. 2 values for 2 channeled memory and 4 four values for 4 channeled memory. We calculated average memory read/write values per channel and used them for ML training. We further obtained mean values of this metric for various memory configurations and reported in Figure 2.

We broadly observed two categories of values for average memory reads per channel which varied by the number of channels used in a memory configuration. Specifically, the average memory reads per channel for a memory with two channels was approximately double ($\approx 4.1 \times 10^7$) the memory reads with four channels ($\approx 2 \times 10^7$). We saw similar observations for average memory writes per channel. For a memory with two channels the average memory writes per channel was ($\approx 4.4 \times 10^7$) and for a memory with four channels the value was ($\approx 2.2 \times 10^7$). DRAM, NVM, and most of the hybrid configurations memories showed this pattern evidently. Few hybrid memory configurations with higher CPU frequencies showed reduction in the values for average memory reads. A hybrid configuration with 2 GHz of CPU frequency, 666 MHz of controller frequency, and 4 channels showed ten times lower value for both average memory reads ($\approx 6.88 \times 10^6$) and writes ($\approx 7.47 \times 10^5$) per channel than the values for other configurations. Other hybrid configurations with similar low values are shown by lighter shades. Form above observations, we can say that the hybrid memory with four channels and with lower CPU frequency showed better performance for graph BFS workload, specifically from the endurance perspective.

From Figures 3(b) and 3(c) and Table I we can see that ML models SVM, RF, and GB including the baseline linear regression, captured this average memory reads and writes per channel well with the $R^2 \approx 1$ while RF's MSE is comparable with that of the baseline linear regression. Moreover, either linear regression or RF can be used for predicting average memory read/writes per channel in co-designing advanced memory architectures. From these observations, we can say that memory reads/writes are the characteristics of a workload. Therefore developing ML models with the data generated from multiple executions of a workload with varying parameters or even different types of workloads would be beneficial for the design space exploration study.

3) Bandwidth: NVMain provides bandwidth (MB/s) for each memory bank. For instance, NVMain executing a memory configuration with 2 channels and 8 banks per channel produces 16 values of bandwidth. We calculated the average of these values to obtain average bandwidth per bank for training ML models. Figure 2 presents mean values of average bandwidth per bank for a memory configuration defined by CPU frequency, controller frequency, and number of channels. Overall, we found that the average bandwidth per bank increased with CPU frequency and controller frequency. Also for this metric DRAM showed higher values than NVM and hybrid memories for graph workload. Moreover, for all memory types the average bandwidth per bank approximately reduced to half when the number of channels doubled for given CPU and controller frequencies.

We can observe in Figure 3(d) that both SVM and GB algorithms predicted the testing data well whereas RF was able to perform well only for lower bandwidth values. We can evaluate this observation in Table I where we can see that SVM outperformed all the other ML models with lower MSE and $R^2 \approx 1$. From these observations we can say that, SVM is a preferred algorithm for predicting average bandwidth per bank for co-designing the advanced memory architectures for graph workloads.

4) Memory Latency: NVMain outputs various latency values such as average latency and total latency. Average latency indicates the number of clock cycles spent after a memory controller initiated a memory request until its completion. This involves operations such as row selection, column selection, and data restoration (in case of DRAM). Total latency includes the queuing delay along with the average latency value. We calculated the average of these two values for ML training, i.e. average latency per channel and average total latency per channel.

Figure 2 provides mean values of these two metrics for various memory configurations based on the first three columns. We found that the hybrid memory performed better over DRAM and NVM for our graph benchmark with low values for average latency per channel. DRAM showed high values and NVM showed average values for the average latency per channel. Overall, the average latency is lower for the memory with four channels than with two channels. The hybrid memory with CPU frequency of 2GHz, controller frequency of 666MHz with four channels showed the lowest value of the average latency for graph benchmark. The average total latency per channel showed a completely reverse trend in which DRAM showed the lowest values compared to NVM and hybrid memories which indicates shorter queuing delay for DRAM. Average total latency per channel was found to be independent of CPU and controller frequency. DRAM with four channels showed slightly lower total latency than with two channels. Whereas, for both NVM and hybrid memory the average total latency per channel increased with the controller frequency.

In Figure 3(e), we can see that all the ML algorithms fits the test data well for average latency per channel for our graph benchmark. Specifically both SVM and RF showed low value for MSE and with $R^2 \approx 1$. However, in Figure 3(f) we can see that SVM outperformed RF and GB resulting in lower MSE and with $R^2 \approx 1$ for average total latency per channel. The R^2 values for both RF and GB were also poor for total latency (refer to Table I). From these observations, we can say that the SVM algorithm is a better choice for predicting average latency and total latency per channel in co-designing the advanced memory architecture for graph workload.

From above results and discussion, we summarize our recommendations for co-design of advanced memory architecture for graph workload as follows:

- We recommend NVM with a controller frequency of 400MHz for better power performance.
- We propose to use hybrid memory with four channels, specifically the one with 2GHz CPU frequency and 666MHz controller frequency for optimal performance for memory reads and writes.
- For better bandwidth performance, we recommend DRAM. However, NVM and hybrid memories also showed comparable performance.
- For optimal performance for average latency we propose using hybrid memory and for total latency we recommend DRAM.
- We recommend using SVM for characterizing the performance metrics bandwidth, power, and latency. We propose linear regression for characterizing memory read and writes for co-designing the advanced memory architectures for the graph benchmark.

V. Concluding Remarks and Future Directions

In this paper, we developed a ML-based design space exploration method to build predictive models for several responses of a hybrid main-memory system. The overarching goal of the project is to build such ML models for various representative workloads. In our earlier work, we experimented with STREAM and HPCG benchmarks [23]. In this work we selected the most representative graph analytics algorithm, i.e. Breadth-First-Search. We believe that the audience of the paper will be interested in the specific numbers of the memory performance metrics for BFS workload.

The work presented in this paper can be further extended in multiple ways. In future, we plan to investigate the generalizability of this work by experimenting with other algorithms in graph analytics, large sized Graph500 benchmarks [24], and different hardware configurations. For this work, we used Gem5 with default configuration in SE mode which uses atomic CPU and atomic memory access. We will extend the current setup for specific CPUs and cache configurations. Specifically, our future work will address the question, *how does the graph size and the type of graph algorithms influence the choice of good parameters for the memory architectures?*

Our methodology has the potential to significantly reduce the computational costs and time associated with simulating memory architectures and optimizing memory performance of graph analytics applications. In our future work, we will utilize more advanced ML methods, such as the transfer learning and semi-supervised learning, to move further beyond the supervised learning domain which strongly depends on the labeled simulated data for training and testing purposes. In addition, we plan to utilize Active Learning (AL) techniques to further enhance our workflow. We will apply intelligent sampling techniques to select the initial labeled training sets for the AL models. This is because an appropriate initial labeled data set would allow the AL models to achieve the supervised performance limits with considerably smaller number of labeled training data, and thus would further improve the efficiency of our DSE approach. We also plan to validate the performance of our AL-based memory-response models with real data.

Acknowledgment

Support for this work was provided by the United States Department of Defense. We used resources of the Computational Research and Development Programs and the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang, "Mambo: A full system simulator for the PowerPC architecture," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 8–12, Mar. 2004.

[2] R. Bedichek, "SimNow: Fast platform simulation purely in software," in *Proc. Hot Chips*, vol. 16, Stanford, CA, Aug. 22-24, 2004, pp. 48–60.

[3] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in *Proc. IEEE 17th Intl Symp. High Perf. Comp. Archit.*, San Antonio, TX, Feb. 12–16, 2011, pp. 406–417.

[4] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. Intl Conf. High Perf. Comp. Net. Storage Analysis*, Seattle, WA, Nov. 12–18, 2011, pp. 1–12.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 17, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[6] M. Poremba and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," in *2012 IEEE Computer Society Annual Symposium on VLSI*, 2012, pp. 392–397.

[7] W. A. Wulf and S. A. Mckee, "Hitting the memory wall: Implications of the obvious," *Computer Architecture News*, vol. 23, pp. 20–24, Mar. 1995.

[8] M. Tolentino, J. Turner, and K. Cameron, "Memory MISER: Improving main memory energy efficiency in servers," *IEEE Transactions on Computers*, vol. 58, no. 3, pp. 336–350, Mar. 2009.

[9] A. Chen, "Emerging research device roadmap and perspectives," in *IEEE Intl Conf. on IC Design Tech.*, Austin, TX, May 28-30, 2014, pp. 1–4.

[10] H. Meuer, E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. (2018, nov) TOP500 Supercomputer Sites. [Online]. Available: https://www.top500.org/

[11] OLCF. (2019) Summit – Oak Ridge Leadership Computing Facility. [Online]. Available: https://www.olcf.ornl.gov/summit/

[12] K. Iwabuchi, S. Ghosh, R. Pearce, M. Halappanavar, and M. Gokhale, "minivite+ metall: A case study of accelerating graph analytics using persistent memory allocator," 2020.

[13] K. Iwabuchi, L. Lebanoff, M. Gokhale, and R. Pearce, "Metall: A persistent memory allocator enabling graph processing," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2019, pp. 39–44.

[14] S. Lim, T. Coy, Z. Lu, B. Ren, and X. Zhang, "Nvgraph: Enforcing crash consistency of evolving network analytics in nvmm systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1255–1269, 2020.

[15] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," *Proc. VLDB Endow.*, vol. 13, no. 10, p. 13041318, Apr. 2020. [Online]. Available: https://doi.org/10.14778/3389133.3389145

[16] L. Dhulipala, "Provably efficient and scalable shared-memory graph processing," Ph.D. dissertation, Intel, 2020.

[17] P. Kumar and H. H. Huang, "Safenvm: A non-volatile memory store with thread-level page protection," in *2017 IEEE International Congress on Big Data (BigData Congress)*, 2017, pp. 65–72.

[18] W. Liu, H. Liu, X. Liao, H. Jin, and Y. Zhang, "Ngraph: Parallel graph processing in hybrid memory systems," *IEEE Access*, vol. 7, pp. 103 517–103 529, 2019.

[19] J. Shun and G. 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.

[20] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 183193. [Online]. Available: https://doi.org/10.1145/2688500.2688507

[21] E. Ipek, O. Mutlu, J. F. Martnez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Intl Symp. Computer Architecture*, Beijing, China, Jun. 21-25, 2008, pp. 39–50.

[22] S. Baek, D. Son, D. Kang, J. Choi, and S. Cho, "Design space exploration of an NVM-based memory hierarchy," in *IEEE 32nd Intl Conf. Computer Design (ICCD)*, Seoul, South Korea, Oct. 19-22, 2014, pp. 224–229.

[23] S. Sen and N. Imam, "Machine learning based design space exploration for hybrid main-memory design," ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 480489. [Online]. Available: https://doi.org/10.1145/3357526.3357544

[24] K. Ueno and T. Suzumura, "Highly scalable graph search for the graph500 benchmark," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, 2012, pp. 149–160.

[25] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their applications*, vol. 13, no. 4, pp. 18–28, 1998.

[26] U. Grömping, "Variable importance assessment in regression: linear regression versus random forest," *The American Statistician*, vol. 63, no. 4, pp. 308–319, 2009.

[27] C. Krauss, X. A. Do, and N. Huck, "Deep neural networks, gradient-boosted trees, random forests: Statistical arbitrage on the s&p 500," *European Journal of Operational Research*, vol. 259, no. 2, pp. 689–702, 2017.

[28] A. Petitet, C. Whaley, J. Dongarra, A. Cleary, and P. Luszczek. (2018) HPL-a portable implementation of the high-performance linpack benchmark for distributed-memory computers. [Online]. Available: http://www.netlib.org/benchmark/hpl/

[29] K. Madduri, "Gtgraph: A suite of synthetic random graph generators," *http://www. cse. psu. edu/˜ madduri/software/GTgraph*, 2006.

[30] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.