

UC Merced

UC Merced Electronic Theses and Dissertations

Title

Memory Management for Big Memory Systems

Permalink

<https://escholarship.org/uc/item/8hs0p4gw>

Author

Ren, Jie

Publication Date

2022

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at <https://creativecommons.org/licenses/by/4.0/>

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, MERCED

Memory Management for Big Memory Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering and Computer Science

by

Jie Ren

Committee in charge:

Professor Dong Li, Chair
Professor Hyeran Jeon
Professor Xiaoyi Lu

2022

Copyright
Jie Ren, 2022
All rights reserved.

The dissertation of Jie Ren is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

(Professor Hyeran Jeon)

(Professor Xiaoyi Lu)

(Professor Dong Li, Chair)

University of California, Merced

2022

TABLE OF CONTENTS

	Signature Page	iii
	Table of Contents	iv
	List of Figures	viii
	List of Tables	xi
	Acknowledgements	xii
	Vita and Publications	xiii
	Abstract	xv
Chapter 1	Introduction	1
Chapter 2	Background	7
	2.1 Big Memory Applications	7
	2.2 Heterogeneous Memory Based Big Memory Systems	8
Chapter 3	Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory	10
	3.1 Introduction	11
	3.2 Preliminary and Related Works	13
	3.2.1 ANNS and Similarity Graphs	13
	3.2.2 Heterogeneous Memory	13
	3.3 HM-ANN	14
	3.3.1 Graph Construction via Top-Down Insertions and Bottom-up Promotions	14
	3.3.2 HM-ANN Graph Search Algorithm	16
	3.3.3 Performance Model-Guided Parameter Selection	18
	3.3.4 Complexity Analysis	19
	3.4 Evaluation	20
	3.4.1 Methodology	20
	3.4.2 Experiment Results	21
	3.4.3 Ablation Studies	23
	3.5 Conclusions	25
Chapter 4	Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning	26
	4.1 Introduction	27
	4.2 Background	30
	4.3 Analysis and Characterization of Main Memory Accesses in DNN	32
	4.3.1 Profiling Framework	32
	4.3.2 Observations and Preliminary Analysis	33
	4.4 Design	35

	4.4.1	Overview	35
	4.4.2	Dynamic Profiling and Data Reorganization	36
	4.4.3	Handling Short-Lived Tensors	37
	4.4.4	Adaptive Layer-Based Migration	38
	4.4.5	Discussions	41
	4.5	Applying Sentinel to GPU	42
	4.6	Implementation	43
	4.7	Experimental Results	45
	4.7.1	Experimental setup	45
	4.7.2	Sentinel on Optane-based HM	45
	4.7.3	Sentinel on GPU-based HM	49
	4.8	Related Work	52
	4.9	Conclusions	53
Chapter 5		Democratizing Billion-Scale Model Training	54
	5.1	Introduction	55
	5.2	Background and Related Work	58
	5.3	Unique Optimal Offload Strategy	61
	5.3.1	DL Training as a Data-Flow Graph	61
	5.3.2	Limiting CPU Computation	62
	5.3.3	Minimizing Communication Volume	63
	5.3.4	Maximizing Memory Savings	64
	5.3.5	A Unique and Optimal Offload Strategy	64
	5.4	ZeRO-Offload Schedule	65
	5.4.1	Single GPU Schedule	65
	5.4.2	Scaling to Multi-GPUs	66
	5.5	Optimized CPU Execution	67
	5.5.1	Implementing the CPU Optimizer	67
	5.5.2	One-Step Delayed Parameter Update	69
	5.6	Evaluation	70
	5.6.1	Evaluation Methodology	70
	5.6.2	Experimental Results	72
	5.7	Conclusions	79
Chapter 6		Enabling Large Dynamic Neural Network Training with Learning-based Memory Management	80
	6.1	Introduction	81
	6.2	Background	85
	6.2.1	Dynamic Neural Networks	85
	6.2.2	Breaking Memory Capacity Wall	85
	6.2.3	Using Machine Learning to Guide Tensor Migration	87
	6.3	Overview	87
	6.4	Design	89
	6.4.1	Design of Input Features	89
	6.4.2	Output of the Learned Model	91
	6.4.3	Light Neural Network Model	93

	6.4.4	Model Training	95
	6.4.5	Runtime Design	95
	6.5	Implementation	97
	6.6	Evaluation	99
	6.6.1	Methodology	99
	6.6.2	Overall Performance	100
	6.6.3	Performance Analysis	101
	6.6.4	Scalability of DyNN-Offload	101
	6.6.5	Construction of Learned Model	102
	6.6.6	Idiom-based Representation	103
	6.6.7	Evaluation of DyNN Model Partition	103
	6.6.8	Impact of Handling Misprediction	104
	6.7	Related Work	104
	6.8	Conclusion	105
Chapter 7		Optimizing Large-Scale Plasma Simulations on Big Memory System . .	106
	7.1	Introduction	106
	7.2	Background	108
	7.3	Performance Characterization	110
	7.3.1	Profiling Results	111
	7.4	Performance Optimization on PM	113
	7.4.1	Static Data Placement	113
	7.4.2	Dynamic Data Placement	115
	7.4.3	Implementation Details	119
	7.5	Evaluation	120
	7.5.1	Experimental Setup	120
	7.5.2	Evaluation Results	121
	7.6	Related Work	125
	7.7	Conclusions	125
Chapter 8		Scalable Page Management for Multi-Tiered Large Memory Systems . .	127
	8.1	Introduction	127
	8.2	Background and Related Work	131
	8.2.1	Multi-Tiered Large Memory Systems	131
	8.2.2	Large Memory Systems	132
	8.2.3	Two-Tiered Heterogeneous Memory	133
	8.3	Overview	133
	8.4	Adaptive Memory Profiling	134
	8.4.1	Adaptive Memory Regions	135
	8.4.2	Adaptive Page Sampling	136
	8.4.3	Profiling Overhead Control	137
	8.5	Holistic Page Management	138
	8.5.1	Which Memory Region to Migrate?	139
	8.5.2	Where to Migrate Memory Regions?	140
	8.6	Adaptive Migration Mechanism	140
	8.6.1	Performance Analysis of Page Migration Mechanism	140

	8.6.2 Adaptive Page Migration Schemes	141
	8.7 Implementation	142
	8.8 Evaluation	143
	8.8.1 Experimental Setup	143
	8.8.2 Overall Performance	144
	8.8.3 Effectiveness of Adaptive Profiling	149
	8.8.4 Effectiveness of Migration Strategy	150
	8.8.5 Effectiveness of Migration Mechanism	151
	8.9 Conclusions	152
Chapter 9	Conclusions and Future Directions	153
	9.1 Conclusions	153
	9.2 Future Directions	155

LIST OF FIGURES

Figure 2.1:	The architecture of heterogeneous memory based big memory systems. . .	8
Figure 3.1:	Query time vs. recall curve in (a) DEEP1B top-1, (b) BigANN top-1, (c) DEEP1B top-100, (b) BigANN top-100, respectively.	22
Figure 3.2:	Query time vs. recall curve with (a)DEEP1M, (b)SIFT1M, and (c)GIST respectively.	23
Figure 3.3:	Comparison of two promotion strategies.	24
Figure 3.4:	HNSW with parallel L0 search.	24
Figure 3.5:	Comparison of techniques in HM-ANN.	24
Figure 3.6:	Comparison of data management methods.	25
Figure 3.7:	Distribution of <i>efSearch</i>	25
Figure 3.8:	Performance with various <i>efSearch</i>	25
Figure 4.1:	An example to show tensor access patterns across layers in ResNet-32. This figure shows the first three layers and the last three layers in ResNet-32. . .	34
Figure 4.2:	Data processing in a TensorFlow operation, <code>nn.conv2d</code>	34
Figure 4.3:	Overview of Sentinel. The white and shadow boxes represent functionality and mechanisms, respectively.	35
Figure 4.4:	Tensor migration based on the migration intervals. “S” and “F” stand for slow and fast memories respectively.	38
Figure 4.5:	Performance (training throughput) variance as we change the migration interval length (MIL). “SP” stands for sweet spot (the optimum migration interval length).	38
Figure 4.6:	Implementation overview.	44
Figure 4.7:	Performance speedup of IAL, AutoTM and Sentinel over slow-memory only. The red horizontal line shows performance of fast-memory only. Performance is normalized by that of slow-memory only.	47
Figure 4.8:	Performance with first-touch NUMA, Memory Mode, AutoTM and Sentinel, normalized by that of first-touch NUMA.	47
Figure 4.9:	Memory access bandwidth during training of ResNet-32.	48
Figure 4.10:	Performance with Sentinel under various sizes of fast memory. The fast memory size is shown as the percentage of peak memory consumption of DNN models. Performance is normalized by that of the fast memory-only.	49
Figure 4.11:	Comparison between peak memory consumption of DNN models and fast memory size for ResNet variants.	49
Figure 4.12:	Performance of UM, vDNN, AutoTM, SwapAdvisor, and Capuchin and Sentinel-GPU, normalized by that of UM. vDNN cannot work for BERT and LSTM.	51
Figure 4.13:	Performance breakdown for vDNN, AutoTM, SwapAdvisor, Capuchin and Sentinel-GPU. “det. MI” stands for “determine an appropriate migration interval length”. Percentage numbers on top of bars are the ratio in terms of execution time of one training step.	52

Figure 5.1:	ZeRO-Offload can be enabled with a few lines of change. The code on left shows a standard training pipeline, while the right shows the same pipeline with ZeRO-Offload enabled.	57
Figure 5.2:	The dataflow of fully connected neural networks with M parameters. We use activation checkpoint to reduce activation memory to avoid activation migration between CPU and GPU.	63
Figure 5.3:	ZeRO-Offload training process on a single GPU.	65
Figure 5.4:	ZeRO-Offload data placement with multiple GPUs	66
Figure 5.5:	Code representing ZeRO-Offload that combines unique optimal CPU offload strategy with ZeRO-powered data parallelism.	66
Figure 5.6:	Delayed parameter update during the training process.	69
Figure 5.7:	The size of the biggest model that can be trained on single GPU, 4 and 16 GPUs (one DGX-2 node).	73
Figure 5.8:	The training throughput with PyTorch, L2L, SwapAdvisor and ZeRO-Offload on a single GPU with a batch size of 512.	73
Figure 5.9:	Training throughput with PyTorch, ZeRO-2, Megatron-LM, ZeRO-Offload without model parallelism and ZeRO-Offload with model parallelism. . . .	75
Figure 5.10:	Comparison of training throughput between ZeRO-Offload and ZeRO-2 using 1–128 GPUs for a 10B parameter GPT2.	76
Figure 5.11:	The training throughput is compared for w/o DPU and w/ DPU to GPT-2. Batch size is set to 8.	77
Figure 5.12:	The training loss curve of unmodified GPT-2, ZeRO-Offload w/o DPU and ZeRO-Offload with DPU.	77
Figure 5.13:	The fine-tuning loss curve of BERT, ZeRO-Offload w/o DPU and ZeRO-Offload with DPU.	77
Figure 5.14:	Comparison of training throughput with enabling offload strategies and optimization techniques step-by-step in ZeRO-Offload.	78
Figure 6.1:	An DyNN example.(a) Implementation of each tree node in DyNN (b) The Tree-FC network where S, PP, NP and VP stand for sentence, preposition phrase, noun phrase, and verb phrase respectively.	86
Figure 6.2:	The overview of DyNN-Offload.	88
Figure 6.3:	An AFM example to show the static structure of the Tree-FC shown in Figure 6.1. (a) A node in DyNN with input and output tensors; (b) AFM representation along with computation in operators.	92
Figure 6.4:	An example of the learned model output. (a) shows an output of the learned model with three execution blocks. (b) shows the dataflow graph of DyNN with a given input, and how the output maps back to the operators.	93
Figure 6.5:	An example of using DyNN-Offload.	98
Figure 6.6:	Performance comparison between existing solutions and DyNN-Offload with seven workloads.	100
Figure 6.7:	Performance breakdown for DyNN-Offload, DTR and unified memory. . . .	101
Figure 6.8:	Scalability evaluation of DyNN-Offload with a variety of model scales in var-Bert.	102
Figure 6.9:	Scalability evaluation of DyNN-Offload in terms of system scales (i.e., the number of GPUs).	102

Figure 6.10:	Evaluation of the effectiveness of the idiom-based representation.	102
Figure 6.11:	The training time with DyNN-Offload and three heuristic solutions to partition DyNNs.	103
Figure 7.1:	The number of memory allocation/deallocation across iterations.	111
Figure 7.2:	Memory bandwidth consumption in major phases.	112
Figure 7.3:	The overview of data management on Optane-based HM.	114
Figure 7.4:	Performance comparison between Memory mode, Optane-only, NUMA first-touch and WarpX-PM.	122
Figure 7.5:	Performance breakdown of main phases of execution to compare the static and dynamic placement with memory mode.	122
Figure 7.6:	Performance comparison between IAL (a state-of-the-art page migration solution for HM) and WarpX-PM.	123
Figure 7.7:	Memory bandwidth consumption in one iteration.	124
Figure 7.8:	Performance with different number of helper threads. “th” is the number of helper threads.	124
Figure 8.1:	Comparison of different memory profiling methods in terms of their effectiveness of identifying frequently accessed pages (hot pages). The profiling overhead is set as 5% of total execution time.	129
Figure 8.2:	An example of multi-tiered memory system.	131
Figure 8.3:	The overview of memory profiling in HM-Keeper.	135
Figure 8.4:	Performance breakdown for migration mechanisms.	141
Figure 8.5:	Performance comparison between existing solutions and HM-Keeper on the Optane-based multi-tiered memory system.	146
Figure 8.6:	Performance comparison between existing solutions and HM-Keeper on the emulated multi-tiered memory system.	146
Figure 8.7:	Breakdown of application execution time.	147
Figure 8.8:	Execution time of VoltDB with different number of client threads.	148
Figure 8.9:	Evaluating HM-Keeper with different large page sizes. We use the SSSP application for evaluation.	148
Figure 8.10:	Evaluation of HM-Keeper on two-tiered HM and comparison with HeMem.	150
Figure 8.11:	Evaluation of the effectiveness of adaptive memory regions (“AMR”), adaptive page sampling (“APS”), and profiling overhead control (“OC”) in the VoltDB execution time.	150
Figure 8.12:	Execution time with setting various of profiling overhead targets when execute voltDB with HM-Keeper.	151
Figure 8.13:	Comparison between flatten- and hierarchical view-based migration. Performance speedup is calculated based on the performance of first-touch NUMA.	151
Figure 8.14:	Performance comparison between Nimble, <i>move_pages()</i> and HM-Keeper in page migration mechanism. The performance speedup is calculated based on the performance of using <i>move_pages()</i>	151

LIST OF TABLES

Table 2.1:	DRAM, HBM, PM comparison	8
Table 3.1:	Indexing time and memory consumption for graph-based methods on billion-scale datasets	22
Table 4.1:	Comparison between existing work on HM management for DNN training.	31
Table 4.2:	Hardware overview of experimental system.	45
Table 4.3:	DNN model for evaluation. “Sen.”, stands for Sentinel.	46
Table 4.4:	Total size of migrated tensors in one training step.	47
Table 4.5:	Maximum Batch Size with vDNN, AutoTM, SwapAdvisor, Capuchin and Sentinel-GPU	52
Table 5.1:	Memory savings for offload strategies that minimize communication volume compared to the baseline.	64
Table 5.2:	Hardware overview of experimental system.	71
Table 5.3:	Model configuration in evaluation.	72
Table 5.4:	Adam latency (s) for PyTorch (PT) and CPU-Adam.	77
Table 6.1:	Distribution of Jaccard distance value for all training samples.	86
Table 6.2:	DyNNs for evaluation.	99
Table 6.3:	Learned model performance with different model complexity. “LM” stands for learned model.	102
Table 7.1:	Compare the memory capacity and simulation scale on supercomputers . .	110
Table 7.2:	The distribution of object size.	112
Table 7.3:	The breakdown of execution time.	112
Table 7.4:	Notation for performance modeling	116
Table 7.5:	Input problems used in evaluation	120
Table 7.6:	Platform Specifications	120
Table 7.7:	Quantifying memory traffic between two NUMA nodes.	124
Table 8.1:	Hardware overview of experimental system.	145
Table 8.2:	Workloads for evaluation.	145
Table 8.3:	The number of memory accesses when using voltDB.	147

ACKNOWLEDGEMENTS

To begin with, I would like to express my deepest thanks to my advisor, Professor Dong Li. He was the key to me having a memorable PhD journey and pursuing a career in academia. He spent incredible time training and mentoring me, guiding me to grow to be a passionate, independent researcher. When I felt lost, he taught me that keep doing good work, and everything else will follow. When I self-doubted, he always believed in my potential and encouraged me never to be afraid of reaching high. I have been very fortunate to have him as my advisor.

Next, I am very grateful to my committee members, Professor Hyeran Jeon and Professor Xiaoyi Lu, for their great support and feedback about my work. And my special thanks to all my mentors and co-authors, especially Dr. Minjia Zhang, Dr. Ivy Peng, Dr. Leo Chunhua Liao, and Professor Harry Xu. It has been my great pleasure working with them. I really enjoyed our meetings and research discussions, which were invaluable research experiences.

In addition, I would like to thank my friend, Dr. Xueting Li. We shared joy, anxiety, frustration, and happiness altogether in the last five years. I am so lucky to have her by my side. I also would like to thank all my labmates from the UC Merced PASA lab. They have been the greatest set of friends and colleagues.

Last but not least, I want to thank my family. I cannot thank my cousin, Fang Ren, enough for her unconditional support. I enjoyed all the beautiful summer days we spent together. I dedicate this dissertation to my parents, Haomin Guo and Zhaohua Ren, who are the best parents anyone could ever ask for. They are my rock in life.

VITA

- 2017 B. S. in Computer Science and Engineering, Beijing Institute of Technology
- 2022 Ph. D. in Electrical Engineering and Computer Science, University of California, Merced

PUBLICATIONS

- Jie Ren**, Dong Xu, Ivy Peng, Jiawen Liu, Kai Wu, Dong Li. “HM-Keeper: Scalable Page Management for Multi-Tiered Large Memory Systems”. A submission to ASPLOS’23.
- Jie Ren**, Shuangyan Yang, Dong Xu, Christian Navasca, Chenxi Wang, Guoqing Harry Xu, Dong Li. “DyNN-Offload: Enabling Large Dynamic Neural Network Training with Learning-based Memory Management”. A submission to ASPLOS’23.
- Jie Ren**, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li and Yuxiong He. “ZeRO-Offload: Democratizing Billion-Scale Model Training”. In 27th USENIX Annual Technical Conference. ATC’21.
- Jie Ren**, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li, “Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning”. In 27th IEEE International Symposium on High-Performance Computer Architecture. HPCA’21.
- Jie Ren**, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li, “Optimizing Large-Scale Plasma Simulations on Persistent Memory-based Heterogeneous Memory with Effective Data Placement Across Memory Hierarchy”. In 35th International Conference on Supercomputing. ICS’21.
- Kai Wu, **Jie Ren**, Ivy Peng and Dong Li. “ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory”. In 19th USENIX Conference on File and Storage Technologies. FAST’21.
- Jiawen Liu, **Jie Ren**, Roberto Gioiosa, Dong Li and Jiajia Li. “Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory.”. In 26th Principles and Practice of Parallel Programming. PPOPP’21.
- Jie Ren**, Minjia Zhang, and Dong Li, “HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory”. In 34th Conference on Neural Information Processing Systems. Neurips’20.
- Jie Ren**, Kai Wu, and Dong Li, “EasyCrash: Exploring Non-Volatility of Non-Volatile Memory for High Performance Computing Under Failures”. In IEEE International Conference on Cluster Computing. CLUSTER’20.
- Ivy Peng, Kai Wu, **Jie Ren**, Maya Gokhale and Dong Li, “Demystifying the Performance of HPC Scientific Applications on Non-Volatile Memory-based Memory Systems”. In International Parallel and Distributed Processing Symposium. IPDPS’20.

Kai Wu, Ivy Peng, **Jie Ren**, and Dong Li, “Ribbon: High Performance Cache Line Flushing for Persistent Memory”. In 29th International Conference on Parallel Architectures and Compilation Techniques. PACT’20.

Jie Ren, Kai Wu, and Dong Li, “Understanding Application Recomputability without Crash Consistency in Non-Volatile Memory”. In Workshop on Memory Centric Programming for HPC. MCHPC’18.

Kai Wu, **Jie Ren**, and Dong Li, “Runtime Data Management on Non-Volatile Memory-based Heterogeneous Memory for Task-Parallel Programs”. In 30th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC’18.

ABSTRACT OF THE DISSERTATION

Memory Management for Big Memory Systems

by

Jie Ren

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California Merced, 2022

Professor Dong Li, Chair

The memory system has been evolving at a fast pace recently, driven by the emergence of large-scale applications and the advance in hardware technology. This trend calls for the birth of big memory systems with extreme heterogeneity, which combines multiple memory technologies with different latency, bandwidth, and capacity to construct main memory. The conventional memory management methods are not adequate to handle the increasing complexity that heterogeneity brings forward, and will fail to deliver the full potential of the new memory. Specifically, the heterogeneity of memory systems brings a substantial disparity in the performance and efficiency, making the decision of which technology to use at what times intricate.

The performance of heterogeneous memory-based big memory systems highly depends on the data locality. By dynamically profiling memory access behaviors, memory management solutions move frequently accessed (or hot) data from slow to fast memory and less frequently accessed (or cold) data from fast to slow memory. However, the decision of how to spread data across all memory components in big memory systems, which data to move, and at what times is not trivial.

This dissertation proposes a series of techniques, spanning from architecture, runtime systems, operating systems, programming models, to applications and algorithms, which are used to efficiently identify *which* data in applications are performance-critical, timely determine *when* to perform data migration, and select *where* to place those data in big memory systems. Specifically, this dissertation contains six common scenarios using big memory systems with big memory applications such as machine learning/artificial intelligence applications, large-scale scientific simulation, and in-memory databases. We propose several software techniques for the efficient use of big memory systems in all scenarios. By doing so, this dissertation identifies bottlenecks in the existing memory management solution, explores the unique characteristics of different types

of applications, and bridges gaps from different system stacks to remedy bottlenecks. Evaluation results based on real-world big memory systems illustrate that with efficient data placement, our solutions outperform existing OSes-based and hardware-based memory management solutions.

Chapter 1

Introduction

The era of big memory systems come upon us. From the application perspective, we see the emergence of machine learning, big data, scientific simulation, and in-memory database applications, which call for large memory capacity because of large datasets and high performance enabled by high memory bandwidth for efficient data processing. From the hardware technology perspective, the emergence of 3D XPoint, through silicon via (TSV) technology and fast interconnect, brings the potential of significantly increasing memory capacity or memory bandwidth while reducing memory production cost. Furthermore, the emergence of GPU-like accelerators brings large performance improvement to data-intensive applications, but imposes high requirements on memory bandwidth and memory capacity. The above trend is calling for the birth of memory systems with extreme heterogeneity, which combines multiple memory technologies to construct big memory systems.

We summarize the features of big memory systems as follow.

- **Memory heterogeneity.** The big memory systems combine multiple memory technologies with differences in latency, bandwidth, cost, and other new design parameters such as persistence and asymmetric read and write performance. The extreme heterogeneity requires to redesign the memory hierarchy to accommodate big memory systems.
- **Huge memory capacity.** Different from traditional memory systems, which have only a few hundreds of gigabytes of DRAM per node, a typical big memory system contains a few terabyte of memory per node. One of the commonly used big memory system is build with Intel Optane DC PM and DRAM. With Intel Optane DC PM, the memory capacity on a single machine can achieve 6TB [83]. The Amazon web service also provides big memory

instances (up to 24 TB per node) for customers to conduct the memory intensive workloads. The huge memory capacity requires memory management techniques must be scalable.

Memory management is one of the most important and challenging tasks in the computer system, which includes memory (de)allocation, data placement and migration, etc. The rapid development of memory techniques has brought new challenges in memory management - the memory system becomes more complicated than ever. To maximize the performance of big memory systems, via proper data tiering, allows for the desired performance levels of the aforementioned classes of applications. To achieve this, software-level [52, 61, 116, 162, 226, 227, 234, 241] and hardware-level [13, 31, 168, 169] approaches in heterogeneous memory (HM) management build the necessary mechanisms to maximize the utility of the fastest available memory component via corresponding dynamic migration of frequently accessed data. The task to identify which data is most appropriate to move and at what times, depending on the available data access information and performance estimates.

A typical memory management (i.e., performance optimization) on HM includes three steps, which are *memory profiling*, *migration strategy* and *migration mechanism*. The first step, *memory profiling*, answers the question to *which* data in applications are performance-critical and need to involve in data migration. The second step, *migration strategy*, answers questions to *when* to perform data migration, and *where* to place those data in HM. The third step, *migration mechanism*, answers the question to *how* to efficiently migrate data.

The unique features in big memory systems makes the memory management becoming particular challenging. We summarize the challenges as follow.

Memory profiling. Big memory systems require high accurate and scalable memory profiling methods. It's difficult to striking the right balance between accurate and overhead when profiling big memory systems. Some existing work [103, 141] explore sampling-based hardware performance counters such as Intel's PEBS and AMD's IBS to track memory accesses. Performance counters count and sample specific events by hardware, which leads to inaccurate profiling results, especially in TB-scale big memory system. Some existing works [78, 95, 104, 248] manipulate specific bits in page table entries (PTEs) to track memory accesses at a per-page granularity. Manipulating PTE provides high accuracy in tracking memory accesses. However, the profiling overhead scales linearly with the number of tracked pages, which takes several seconds to track millions of pages (i.e., TB-scale, which is commonly seen in bit memory systems). Such profiling method is too slow to respond to time-changing access patterns. In this dissertation, we proposal techniques which utilize applications' domain knowledge to reduce profiling overhead

in Chapter 3, 4, 5, and 7. We demonstrate machine learning can be used to predict the memory access patterns in dynamic neural networks (DyNN) to avoid expensive memory profiling in Chapter 6. We also introduce a high-accurate, lightweight memory profiling method for TB-scale multi-tiered memory based on adaptively merging/splitting memory regions, described in Section 8.

Migration strategy. Migration strategy needs to make data migration decision based on memory profiling results. It should timely capture hot data without violating the capacity of fastest (or fast) memory. Existing work [98, 234] leverage caching algorithm such FIFO, LRU as migration strategy. Caching algorithm is not suitable for big memory systems because following two reasons. First, the fastest (or fast) memory component in big memory systems contains hundreds of gigabytes. Maintaining FIFO or LRU list for fastest (or fast) memory component is too expensive to make data migration decision timely. Second, big memory systems usually contains multiple memory tiers. A single FIFO or LRU list can not provide enough information to make migration decision due to multiple migration destination. The dissertation uses a greedy algorithm to fetch data into fast memory aggressively in Chapter 3; proposes a lightweight performance model to decide whether performs data migration based on memory component bandwidth and latency in Chapter 7, and proposes a migration strategy for multiple memory tier based on memory access heuristic in Chapter 8.

Migration mechanism. Data migration is not free. Data migration overhead depends on performance features of source and destination memory components, as well as data structure and data size of migrated data. Minimizing the data migration overhead is critical to obtain performance gain from memory management. Existing work [234] leverages concurrent migration and bi-direction migration to reduce the huge page migration overhead in Linux. The dissertation discuss how to maximize the overlap between data migration and computation using performance model in Chapter 4 and 7. In Chapter 8, we also proposal a novel data migration mechanism to optimize read intensive data migration performance.

The above challenges in existing memory management runtime motivate the development of our techniques that can break the trade-offs in big memory systems management and improve the runtime system efficiency for modern big memory applications. This dissertation consists of a series of techniques, spanning operating systems (OSes), runtime systems, applications, and algorithms, aiming to address above challenges, optimizing away runtime system inefficiencies, and thus, significantly improve the efficiency of big memory systems while making them easier for users to deploy and manage their applications. In particular, the contributions of this dissertation

are:

- **Contribution 1.** *HM-ANN* [181], a novel approximate nearest neighbor search (ANNS) algorithm which breaks the fundamental tradeoff in ANNS algorithms between query latency and accuracy and enables fast and highly accurate billion-scale ANNS on HM based big memory systems.

The state-of-the-art ANNS algorithms face a fundamental tradeoff between query latency and accuracy, because of limited main memory capacity: To store indices in main memory for fast query response, they have to limit the number of data points in DRAM or store compressed vectors, which hurts search accuracy. The emergence of HM brings opportunities to largely increase memory capacity and break the above tradeoff. However, HM consists of both fast (but small) memory and slow (but large) memory, and using HM inappropriately slows down query time significantly. To make best utilization of both fast and slow memory in HM, *HM-ANN* takes both memory and data heterogeneity into consideration and enables billion-scale similarity search on a single node. *HM-ANN* provides 95% top-1 recall in less than 1ms on billion-scale datasets, and is 2x-5.8x faster than state-of-the-art.

- **Contribution 2.** *Sentinel* [178], a software runtime system for Tensorflow that automatically optimizes tensor placement in static deep neuron networks (DNN) training on HM.

Memory capacity is a major bottleneck for training deep DNN. HM combining fast (e.g., GPU device memory) and slow memories (e.g., CPU host memory) provides a promising direction to increase memory capacity. However, HM imposes challenges on tensor migration and allocation for high performance DNN training. Prior work unnecessarily causes tensor migration due to page-level false sharing, and wastes fast memory space. To address above problems, we design *Sentinel*, which coordinates operating system (OS) with runtime-level profiling dynamically, and enables co-allocating tensors with similar lifetime and memory access frequency into the same pages to improve tensor movement efficiency. With the above optimizations, *Sentinel* successfully avoids out-of-memory (OOM) issues for large models on GPU, and outperforms five state-of-the-art memory management solutions for DNN training.

- **Contribution 3.** *ZeRO-Offload* [179], a novel heterogeneous deep learning (DL) training technology that makes large transformer-based model training accessible to everyone.

Training extremely large models such as natural language processing (NLP) models with billions of parameter is challenging. It often requires refactorization of the models and the accesses to prohibitively expensive GPU clusters. *ZeRO-Offload* utilizes both memory and computation resources in host CPU to enable large DL model training with limited GPU resources. Specifically, *ZeRO-Offload* places memory-consuming tensors such as optimizer states on CPU memory, and computation-intensive tensors such as parameters on GPU memory. *ZeRO-Offload* enables 10x bigger model training on a single GPU and achieves near-linear speedup on up to 128 GPUs.

- **Contribution 4.** *DyNN-Offload*, a memory management system to train dynamic neural networks (DyNN) with limited GPU capacity.

Managing tensors in DyNN to save GPU memory is challenging, because the dynamic model structure of DyNN leads to input-dependent tensor access patterns. *DyNN-Offload* proposes a learned approach (using a neural network) to increase the predictability of tensor accesses to facilitate memory management. *DyNN-Offload* enables fast inference while providing high prediction accuracy of the learned model. Specifically, *DyNN-Offload* reduces input feature space and model complexity based on a new representation of DyNN, and converts the hard problem of making predictions for individual tensors or operators into a simpler problem of making predictions for a group of operators. We implement *DyNN-Offload* with cross-platform machine learning library onnxruntime. *DyNN-Offload* outperforms state-of-the-art solutions by 2%-50% in terms of training time with the same GPU memory capacity and enables 8x larger model training without out of memory.

- **Contribution 5.** *WarpX-PM* [177], a automatic data placement solution for particle-in-cell (PIC) method on HM-based big memory systems.

The PIC method is an important model that uses computational particles to simulate plasma particles, such as electrons and protons. High-fidelity PIC simulations often use billions and even trillions of particles, which requires high memory capacity. I explored the usage of HM to enable large-scale plasma simulations at unprecedented scales on a single machine. By analysing the performance of PIC simulation in detail and designed a novel dynamic data placement strategy, we implemented the strategy in a DOE mission-critical application, WarpX. *WarpX-PM* accelerates the execution of WarpX on HM by over 60% (compared with the case of no management).

- **Contribution 6.** *HM-Keeper*, an application-transparent page management system that

supports the efficient use of multi-tiered big memory systems.

Multi-terabyte big memory systems are often characterized with more than two memory tiers for large memory capacity and high performance. Those tiers include slow and fast memories with different latencies and bandwidths. Making effective, transparent use of the multi-tiered large memory system requires a page management system, based on which the application can make the best use of fast memories for high performance and slow memories for large capacity. However, applying existing solutions to multi-tiered large memory systems has fundamental limitation because of non-scalable, low-quality memory profiling mechanisms and unawareness of rich memory tiers in page migration policies. To address the above problems, *HM-Keeper* is designed based on two principles: (1) The memory profiling mechanism must be adaptive based on spatial and temporal variation of memory access patterns. (2) The page migration must employ a holistic design principle, such that any slow memory tier has equal opportunities to directly use the fastest memory. *HM-Keeper* largely outperforms existing page management solutions on large memory systems. Memory-consuming applications such as in-memory databases, billion-scale graph processing, and many other big data applications can greatly benefit from *HM-Keeper*.

The remainder of this dissertation document is organized as follows.

Chapter 2 includes background descriptions for the emerging of big memory applications 2.1, the emerging of advanced hardware techniques 2.2. Chapter 3 introduces *HM-ANN*, a efficient billion-point nearest neighbour search on HM-based big memory systems. Chapter 4 describes *Sentinel*, a efficient tensor migration and allocation solution on HM for deep learning. Chapter 5 proposes techniques used in *ZeRO-Offload* to democratizing billion-scale model training. Chapter 6 introduces a learning based approach to predict tensor accesses in DyNN called *DyNN-Offload*. Chapter 7 discusses *WarpX-PM*, which contains optimizing large-scale plasma simulations on HM with effective data placement across memory hierarchy. Chapter 8 introduces a scalable page management solution for multi-tiered big memory systems called *HM-Keeper*. Finally, we conclude in Chapter 9 with some exciting future directions.

Chapter 2

Background

2.1 Big Memory Applications

As application data sizes ever exploding, big memory applications are emerging. Big memory applications are used to solve big problems in math and science. The memory access patterns in big memory applications typically are complex and irregular. Traditional memory management techniques fail to scale in the necessary capacities and speeds to accelerate modern analytics. We summarize the commonly used big memory applications as follow.

Machine learning and artificial intelligence applications. Machine learning and artificial intelligence applications, such as deep neural networks (DNN), have been shown preliminary success in many field. The memory wall exist in both training and inference DNN models. For example, NLP model size has been increased 200x in last three years. Existing works try to reduce memory consumption and break the memory wall of deploying ML/AI applications, such as using low-precision tensors [236], distributed training [72, 115, 173, 195], tensor redundancy removal [172], tensor migration on heterogeneous memory [77, 79, 158, 178, 179], and tensor rematerialization [38, 55, 88, 100, 196].

Large-scale scientific simulations. Large-scale scientific simulations are widely used to understand physics, chemistry, and biology. Scientific simulation can be extremely memory-consuming. For example, in plasma simulation [30], the simulation quality depends on the number of particles used in the simulation. To enable high-resolution scientific simulation, hundreds of supercomputer nodes are used simultaneously [208, 210]. The large capacity of persistent memory (PM) provides the opportunity to enable extra-scale scientific simulation. However, memory management for scientific simulation is challenging due to its extremely complex data

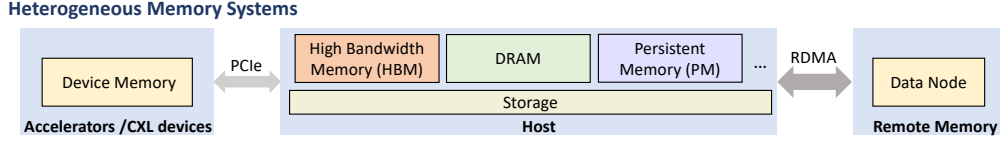


Figure 2.1: The architecture of heterogeneous memory based big memory systems.

access patterns.

In-memory database. In-memory database are widely used in cases where the response time is critical, such as real-time bidding and caching. Existing works deploy in-memory database on non-volatile memory DIMM (NVDIMM) to utilize NVDIMM’s high memory capacity [1, 2, 8, 22, 46, 130, 213]. The data access patterns of in-memory database are usually irregular, depending on the data structure used in the database and database access distribution. The unique challenge in memory management at the operating system (OS) level for in-memory database comes from memory management techniques such as buffers commonly used in-memory database at application-level, which can conflict with OS-level memory management. To get the best performance, memory management solutions must be coordinate at all system stacks.

2.2 Heterogeneous Memory Based Big Memory Systems

The emergence of advanced hardware technologies such as 3D XPoint, through silicon via (TSV) technology and fast interconnect, brings the potential of significantly increasing memory capacity or memory bandwidth while reducing memory production cost. We introduce real-world memory devices which as commonly used to build heterogeneous memory based big memory systems as follow.

Table 2.1: DRAM, HBM, PM comparison

	DRAM	High Bandwidth Memory (TSV)	Persistent Memory (3D-Xpoint)
Bandwidth	1x	2x	0.1 - 0.25x
Latency	1x	0.75x	4 - 8x
Capacity	limited	very small	large
\$/GB	1x	~4x	0.3x

Figure 2.1 shows the architecture of heterogeneous memory based big memory systems. The host memory system contains High Bandwidth Memory (HBM), traditional DRAM, and Persistent Memory. The host memory system can access data in accelerators’ memory or fabric attached memory such as CXL device through PCIe. The host memory system can also access

remote memory node through RDMA. Different memory components in figure 2.1 has significant difference in terms of memory access latency, bandwidth, capacity, and production cost. Table 2.1 summarize the difference between HBM, DRAM and PM.

Chapter 3

Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory

The state-of-the-art approximate nearest neighbor search (ANNS) algorithms face a fundamental tradeoff between query latency and accuracy, because of small main memory capacity: To store indices in main memory for fast query response, They have to limit the number of data points or store compressed vectors, which hurts search accuracy. The emergence of heterogeneous memory (HM) brings opportunities to largely increase memory capacity and break the above tradeoff: Using HM, billions of data points can be placed in main memory on a single machine without using any data compression. However, HM consists of both fast (but small) memory and slow (but large) memory, and using HM inappropriately slows down query time significantly. In this work, we present a novel graph-based similarity search algorithm called HM-ANN, which takes both memory and data heterogeneity into consideration and enables billion-scale similarity search on a single node without using compression. On two billion-sized datasets BIGANN and DEEP1B, HM-ANN outperforms state-of-the-art compression-based solutions such as L&C [51] and IMI+OPQ [50] in recall-vs-latency by a large margin, obtaining 46% higher recall under the same search latency. We also extend existing graph-based methods such as HNSW and NSG with two strong baseline implementations on HM. At billion-point scale, HM-ANN is 2X and 5.8X faster than our HNSW and NSG baselines respectively to reach the same accuracy.

3.1 Introduction

Efficient billion-scale nearest neighbor search has become a significant research problem [23, 24, 90, 93], inspired by the needs of machine learning based applications. Since the number of entities (images, documents, etc) grows enormously fast, it becomes challenging to find correspondences in large datasets when there is a requirement for real-time responses (e.g., in several milliseconds). Exhaustive search is infeasible at billion-point scales, because it is extremely computational demanding. Hence, practitioners resort to indexing structures that perform the approximate nearest neighbor search (ANNS) by restricting a query to search only a subset of the dataset that includes the desired neighbors [29, 68, 108]. Among those ANNS, it has been demonstrated that similarity graphs, such as Hierarchical Navigable Small World (HNSW) [128] and Navigating Spread-out Graph (NSG) [58], obtain superior performance relative to tree structure based [28, 29, 136, 237], locality sensitive hashing (LSH) based [63], and inverted multi-index (IMI) based [108] approaches, and they overall provide the best-in-class latency-vs-accuracy trade-off on most public benchmark datasets.

While obtaining good search speed and accuracy, one major limitation of existing similarity graphs is that they are very memory consuming and easily run out of memory with a few hundred millions of vectors. When the dataset becomes too large to fit on a single machine, the compressed representations of the database points are used, such as Hamming codes [143] and product quantization [51, 59, 89, 94, 142]. However, the performance of these methods deteriorates rapidly at higher recall targets, because they calculate approximate distance based on compressed vectors instead of on the original data vectors. Douze et. al. [51] propose Link-and-Code (*L&C*), which combines a similarity graph with quantized nodes and exploits neighbor nodes to refine the estimation of distance. However, this approach still works poorly at high recall targets. In [198], the authors explore slow storage to achieve billion-scale ANNS in a single machine. However, this approach is based on a fundamental assumption that the persistent media such as SSD is several orders of magnitude slower than DRAM. Based on this assumption, data accesses to the persistent media during search should be zero. As a result, it maintains a copy of compressed data in memory with product quantization [198], which results in loss of in-memory search quality. It then preforms a re-ranking using full-precision coordinates stored on SSD, using block-level data accesses but with expensive SSD accessing time.

In this work, we present a fast and accurate approximate nearest neighbor search algorithm for extremely large scale ANN search, called HM-ANN, which is built on top of Heterogeneous Memory. Heterogeneous Memory (HM) combines cheap, slow but extremely large memory with

expensive, fast but small memory (e.g., traditional DRAM) to achieve a good balance between production cost, memory performance and capacity. The emergence of HM brings opportunities to significantly improve ANNS. Because of the large memory capacity, HM can use full-precision vectors with accurate distance computation. Since memory access latency/bandwidth of slow memory component in HM is much faster than slow storage such as SSD, it is possible to occasionally access data in slow memory during search without paying expensive cost of data accesses. That being said, releasing full performance potential of HM for ANNS is challenging. Although the slow memory such as PMM performs $\sim 80X$ times faster than SSD, it is still $\sim 3X$ slower than DRAM in terms of random access latency. Therefore, a naive data placement strategy can hurt the search efficiency badly. It then raises the following research question: can we leverage HM for ANNS to achieve both high search accuracy and low search latency, especially when the dataset cannot in DRAM (fast memory)? Specifically, the algorithm should have a clear advantage over the state-of-the-art ANNS solutions.

HM-ANN enables fast and highly accurate billion-scale ANNS on HM. In particular, we make the following contributions. (1) We present a fast and accurate billion-scale nearest neighbor search solution on a single node without compression. Specially, we generalize the HNSW construction algorithm to have a top-down insertion phase and a bottom-up promotion phase. The top-down phase creates navigable small world graph as the bottom-most layer, which is also the largest, placed to the slow memory; The bottom-up promotion phase promotes pivot points from the bottom layer graph to form upper layers that are placed in the fast memory, which allows most search accesses to happen in fast memory without losing much accuracy. (2) We explore memory management techniques such as dynamic migration to prefetch to-be-accessed data from slow memory to fast memory and parallel search to reduce search time in slow memory. (3) We introduce a performance model to select search-related hyperparameters that satisfy search time and recall constraints. (4) We conduct extensive evaluation and show that on two billion-scale datasets, HM-ANN provides 95% top-1 recall in less than one millisecond; HM-ANN outperforms state-of-the-art compression-based solutions such as L&C [51] and IMI+OPQ [50] in terms of recall-vs-latency by a large margin, getting 46% higher recall under the same search latency budget; Since NSG and HNSW have never been scaled up to a billion vector on a single machine, we create two strong baselines for them: using first-touch NUMA and hardware-managed caching, respectively. Our results show that for 95% top-1 recall, HM-ANN outperforms the baselines by 2X-5.8X in terms of search latency.

3.2 Preliminary and Related Works

3.2.1 ANNS and Similarity Graphs

Similarity graphs like HNSW [128] and NSG [58] have demonstrated superior performance with polylogarithmic search and graph construction complexity for ANNS [54, 112, 198]. Take HNSW as an example, which consists of multiple layers. The bottom-layer (L0) contains all database elements, and the above layers are randomly selected, nested subsets of database elements. The sizes of the layers follow a geometric progression. During the graph construction phase, HNSW connects elements in each layer based on the closeness relationship. The connections of an element consist both long-range links and short-range links to establish the small world properties. HNSW constrains the length of the neighbors list of each element by a parameter M . HNSW starts the search at the top layer, and performs a *1-greedy search* until it reaches the nearest neighbor of the query in that layer. That node is then used as an entry point in the next layer to start search again. At the bottom layer L0, which contains all elements, HNSW performs a *best-first beam search* to get the final candidates. HNSW uses a parameter *efSearch*, which decides the candidate queue length, to control search time vs. accuracy trade-off. Despite their outstanding performance, similarity graphs are memory-consuming. For example, for the Deep1B [23] dataset, they require 384 bytes per vector, which translates to >350 GB DRAM when including all overheads of data structures, causing out-of-memory failure. Therefore, existing work mostly evaluate their solutions with a few millions vectors [58, 128].

3.2.2 Heterogeneous Memory

Heterogeneous memory (HM) is emerging. It combines multiple memory components to construct main memory. HM is typically composed of a high-capacity memory technology such as non-volatile memory (but high memory access latency) and a high-performance memory technology (with limited memory capacity) such as DRAM. To make HM performance close to that of DRAM-only, previous work focuses on hardware- [13, 31, 168, 169, 215] and software-based [52, 116, 125, 180, 223, 226, 227] solutions to manage data placement on HM. Optane PMM and DRAM are commonly used to build HM. With PMM, the memory capacity on a single machine can achieve 6TB [87]. However, the latency and bandwidth of PMM is only 1/3 and 1/6 of DRAM. There are two operating modes for PMM, *Memory Mode* and *App-direct Mode*. In Memory Mode, DRAM works as a hardware-managed cache to PMM. Running the application in this mode does not require application modifications. App-direct Mode allows the programmer to explicitly control memory accesses to PMM and DRAM. HM-ANN works in App-direct Mode

and outperforms Memory Mode in billion-scale dataset search (Section 3.4).

3.3 HM-ANN

The design of HM-ANN generalizes HNSW, whose hierarchical structure naturally fits into HM. Elements in upper layers consume a small portion of the memory, making them good candidates to be placed in fast memory (small capacity); The bottom-most layer has all the elements and has the largest memory consumption, which makes it suitable to be placed in slow memory. Unlike HNSW, where the majority of search happens in the bottom-most layer, elements in upper layers now have faster access speed, so it is a reasonable strategy to increase the access frequency of upper layers. On the other hand, since accessing L0 is slower, it is preferable to have only a small portion of it to be accessed by each query. The key idea of HM-ANN is therefore to build high-quality upper layers and make most memory accesses happen in fast memory, in order to provide better navigation for search at L0 and reduce memory accesses in slow memory.

Notations. In the rest of the paper, we let V denote the dataset with $N = |V|$ to build the graph; we refer the graph in the layer $i \in \{0, 1, \dots, l\}$ of HM-ANN as $G_i = (V_i, E_i)$ where V_i is the vertex set and E_i is the edge set. We refer N_i as the number of elements in the layer i , and we have $N_i = |V_i|$. Because L0 contains all the elements in database, we have $V_0 = V$ and $N_0 = N$. Based on the hierarchical structure of HM-ANN, we have $V_i \subsetneq V_{i-1}$. Similar to the existing effort [128], we introduce M_i as the maximum number of established connection for each point v in the layer i . For $v \in V$, we let $D(v)$ denote the degree of node v , and $D(v) = \sum_{u \in V} m(v, u)$ where $m(v, u) = 1$ if there exists a link between node v and node u .

3.3.1 Graph Construction via Top-Down Insertions and Bottom-up Promotions

We generalize the HNSW construction algorithm to include two phases: a top-down insertion phase and a bottom-up promotion phase (Alg. 1).

Top-down insertions. The top-down insertion phase is the same as HNSW (Line 1 in Algorithm 1), where we incrementally build a hierarchical graph by iteratively inserting each vector v in V as a node in G . Each node will generate up to M (i.e., the neighbor degree) out-going edges. Among those, $M - 1$ are short-range edges, which connect v to its $M - 1$ nearest neighbors according to their pair-wise Euclidean distance to v . The rest is a long-range edge that connects v to a randomly picked node, which may connect other isolated clusters. It is theoretically justified that graphs (e.g., L0) constructed by inserting these two types of edges guarantees to have the small world properties [58, 128, 221].

Bottom-up promotions. The goal of the second phase is to build a high-quality projection of L0 elements into the layer 1 (L1), such that search in L0 can find true nearest neighbours of the query with only a few number of hops. Ideally, HM-ANN wants to achieve the goal that performing 1-greedy search in L0 is sufficient to achieve high recall, so that the slowdown caused by accessing the slow memory is minimal. A straightforward way to project the L0 elements into L1 is to randomly select a subset of elements in L0 to be L1, similar to what HNSW already does to build upper layers. However, we observe that such an approach leads to poor index quality. As a result, many searches end up happening in L0 (slow memory), causing long search latency.

Algorithm 1: HM-ANN Graph Construction Algorithm.

Input: vector set V , vector dimension d , number of established connection M , size of dynamic candidate list $efConstruction$

Output: Multi-layer graph HM-ANN

Parameters : # of nodes in layer i N_i , HM-ANN layer depth l

```

1  build graph  $hns w \leftarrow HNSW(V, d, M, efConstruction)$ ;
2  for  $v$  in  $V$  do
3  [  $D[v] \leftarrow$  the degree of  $v$  as in zero-layer L0;
4  sort  $D$  for descending order ;
5  remove nodes in layer 1 to  $l$  ;
6   $ep \leftarrow$  get the highest degree node  $v$  in  $D(v)$  ;
7  for  $v$  in  $V$  in  $D(v)$  descending order do
8  [   for  $i \leftarrow l \dots 1$  do
9  [   [   if  $N_i == 0$  then
10 [   [   [   // layer  $i$  is full
11 [   [   [    $W \leftarrow search\_layer(v, \{ep\}, ef = 1, i)$ ;
12 [   [   [    $ep \leftarrow$  get nearest vector from  $W$  to  $v$ ;
13 [   [   [   else
14 [   [   [   [   // add  $v$  in layer  $i$  to 1
15 [   [   [   [   for  $j \leftarrow i \dots 1$  do
16 [   [   [   [   [    $W \leftarrow search\_layer(v, \{ep\}, efConstruction, j)$  ;
17 [   [   [   [   [    $neighbors \leftarrow$  heuristic select  $M_i$  nodes from  $W$  in layer  $j$  ;
18 [   [   [   [   [   add bidirectional connections from  $neighbors$  to  $v$  at layer  $j$ ;
19 [   [   [   [   [   shrink connections if  $\exists q \in neighbors$  and  $D_{out}(q) > M_i$ ;
20 [   [   [   [   [    $N_j = N_j - 1$ ;
21 [   [   [   [   [   beark;

```

HM-ANN uses a *high-degree promotion strategy* (Lines 7-19 in Algorithm 1). This strategy promotes elements with the highest degree in L0 into L1. From the layer i ($i \geq 2$) to $i + 1$, HM-ANN promotes high-degree nodes to upper layer with a promotion rate of $1/M$, where M is

the maximum number of neighbors for each element (i.e., $M_i = M$, where $i = 2..l$). The similar promotion rate setting is used in HNSW [128] and typical skip list [165].

HM-ANN increases search quality in $L1$ by promoting more nodes from $L0$ to $L1$ and setting the maximum number of neighbors for each element in $L1$ to $2 \times M$ (i.e., $M_1 = 2 \times M$). The number of nodes in upper layers (N_i , where $i = 1..l$) is decided by available fast memory space. Excluding the fast memory space for dynamic migration (discussed in Section 3.3.2) and data structure used for search (e.g, the visited elements set VE in Algorithm 3), the remaining fast memory space is used for storing data and links for each node. Section 3.3.4 quantifies memory usage in each layer, from which we can calculate N_i for each layer.

The high-degree promotion strategy is based on the following observation. The hub nodes of the graph at $L0$ are those nodes with a large number of connections (i.e., high degree). In the small world navigation algorithm, a higher degree node provides better navigability [26]. Most of the shortest paths between nodes flow through hubs. In other words, the average length of the navigation path (i.e., number of hops) is the smallest, when the adjacent node with the highest degree is selected as the next hop. By promoting the high-degree nodes, the resulting $L1$ layer allows HM-ANN to effectively reduce the number of search in $L0$, compared with the random promotion strategy.

3.3.2 HM-ANN Graph Search Algorithm

Algorithm 2: HM-ANN K-NN-Search

Input: multi-layer graph HM-ANN, query element q , number of nearest neighbour to return K

Output: K nearest elements to q

Parameters : size of the dynamic candidate list in layer 1 and 0 as $efSearch_{l1}$ and $efSearch_{l0}$ respectively

```

1  $ep \leftarrow$  entry point of HM-ANN;
2  $L \leftarrow$  level of  $ep$  ;
3  $W \leftarrow \emptyset$ ;
4 for  $i \leftarrow L..2$  do
5    $W \leftarrow$  search_layer( $q, \{ep\}, ef = 1, i$ );
6    $ep \leftarrow$  get nearest element from  $W$  to  $q$ ;
7  $W \leftarrow$  search_layer( $q, \{ep\}, efSearch_{l1}, 1$ );
8  $W \leftarrow$  search_layer( $q, W, efSearch_{l0}, 0$ );
9 return  $K$  nearest elements from  $W$  to  $q$ 
```

Fast memory search. The search in fast memory begins at the entry point in the top layer and then performs 1-greedy search from the top layer to the layer 2, which is the same as in HNSW, as

shown in Algorithm 2. To narrow down the search space in L0, HM-ANN performs the search in L1 with a search budget controlled by $efSearch_{L1}$ by using Algorithm 3. $efSearch_{L1}$ defines the size of dynamic candidate list in L1. Those candidates in the list are used as entry points for search in L0 (HNSW uses just one entry point), in order to improve search quality in L0. We provides algorithm details in the appendix.

Parallel L0 search. In L0, HM-ANN evenly partitions the candidates from searching L1 and uses them as entry points to perform *parallel multi-start 1-greedy search* with Thr threads in parallel as shown in Algorithm 3. The top candidates from each search are collected to find the best candidates. Parallel search makes best use of memory bandwidth and improves search quality without increasing search time. Thr is determined by peak memory bandwidth constrained by hardware divided by memory bandwidth consumption by one thread, which is easy to calculate.

Algorithm 3: HM-ANN Search Layer

Input: query vector q , enter points set EP , number of nearest neighbors to query q to return ef , layer number l

Output: ef nearest vectors to q

Parameters : # of threads Thr , set of visited elements VE, set of candidates C , dynamic list of found nearest neighbors W

```

1   $Thr = \min(Thr, |EP|)$ 
2  partition  $EP$  into  $EP_i, i \leftarrow Thr - 1 \dots 0$ 
3  do in parallel
4  |    $VE_t \leftarrow EP; C_t \leftarrow EP_t; W_t \leftarrow EP$ 
5  |   while  $|C_t| > 0$  do
6  |   |   if  $\min\_dist(q, C_t) > \max\_dist(q, W_t)$  then
7  |   |   |   break;
8  |   |   evaluate neighbors of  $c \in C_t$ 
9  |   |   update  $VE_t$  and  $W_t$ 
10 merge  $W_i$  into  $W, i \leftarrow Thr - 1 \dots 0$ 
11 return  $ef$  nearest vectors from  $W$  to  $q$ 

```

Different from the SSD-based ANNS [198, 245], the data in slow memory in HM-ANN can be directly accessed by processors, and there is no duplication between fast and slow memories. However, due to high latency and low bandwidth of slow memory, HM-ANN should still make memory accesses in fast memory as many as possible. HM-ANN implements a software-managed cache in fast memory to prefetch data from slow memory to fast memory before the memory access happens. In particular, HM-ANN reserves a space in fast memory (~ 2 GB) called *migration space*. When searching L1, HM-ANN asynchronously copys neighbor

elements of those candidates in $efSearch_{L1}$ and the neighbor elements' connections in L1 from slow memory to the migration space in fast memory. When the search in L0 happens, there is already a portion of to-be accessed data placed in fast memory, which leads to shorter query time.

3.3.3 Performance Model-Guided Parameter Selection

The overall search quality of HM-ANN is related to the choice of $efSearch$ at L1 (i.e., $efSearch_{L1}$) and $efSearch$ at L0 (i.e., $efSearch_{L0}$), which controls the number of distance computation happens in fast memory and slow memory, respectively. To achieve a low query latency, ideally we would like $efSearch_{L0}$ to be as small as possible, such as 1-greedy search ($efSearch_{L0} = 1$). However, although searching L1 narrows down the L0 search into a small local region, to have a high search quality requires that $efSearch_{L0}$ can not be too small, because the nearest neighbors not included L1 and are not visited in L0 are definitely lost. Given the large search space of $efSearch_{L1}$ and $efSearch_{L0}$, it is preferable to have a systematic way to do parameter selection. This section provides a performance model for HM-ANN, with an eye towards being able to set $efSearch_{L1}$ and $efSearch_{L0}$ properly to meet the goal of having low response time and high accuracy.

Response time constraint. To provide interactive service, the search latency must be lower than a response time limit. In HM-ANN, we model the search latency as $T = T_{L1*} + T_{L0}$, where T_{L1*} models search time in L1 and above, which is primarily dominated by search in L1, and T_{L0} models search time in L0. The average query time at a layer is bounded by $efSearch \times C \times T_{DC}$, where $efSearch$ is the size of dynamic candidate list in the layer and can be viewed as the beam length in the best-first beam search; C is the average number of distance computations per beam before finding the nearest neighbor at a layer; T_{DC} is the execution time to calculate a pair-wise distance.

T_{DC} is a constant and can be measured offline on both fast memory ($T_{DC_{fast_mem}}$) and slow memory ($T_{DC_{slow_mem}}$). C is calculated by $C = \#steps \times DC_per_step$, which is a multiplication of the average number of steps before we reach the nearest neighbor ($\#steps$) and maximum number of distance computation per step (DC_per_step). $\#steps$ in a layer is bounded by a constant [128] based on the theory of Delaunay graph and is independent of the dataset size; DC_per_step is bounded by the maximal out-degree M . When modeling search time in L0, we consider the effect of parallel search with a parallel degree Thr . For the execution time, we therefore have:

$$\begin{aligned}
T &= T_{L1^*} + T_{L0} \\
&= efSearch_{L1} \times C \times T_{DC_{fast_mem}} + \left\lceil \frac{efSearch_{L1}}{Thr} \right\rceil \times efSearch_{L0} \times C \times T_{DC_{slow_mem}} \\
&\leq search_time_constraint
\end{aligned} \tag{3.1}$$

Satisfy both response time and accuracy constraint. Beyond response time constraint, high accuracy is clearly also important for high-quality ANNS, because otherwise users will not be able to find what they are looking for. In practice, the accuracy of search must be higher than an accuracy target θ . Therefore, for a given HM-ANN graph, HM-ANN first applies Equation 3.1 to analytically get a set of candidate $(efSearch_{L0}, efSearch_{L1})$ pairs that satisfy the response time constraint. This step often significantly reduces the search space to only a small set of configurations.

Among those candidate pairs, HM-ANN uses a learning query set randomly sampled to measure the expected accuracy $\mathbb{E}(\theta)$, with $efSearch_{L1} \geq 1$, and $efSearch_{L0} \geq 0$ as constraints. HM-ANN then chooses those configurations that satisfy $\mathbb{E}[\theta] \geq \theta$. Finally, HM-ANN uses grid search to choose the configuration that leads to the shortest query time.

3.3.4 Complexity Analysis

Search complexity. HM-ANN constructs each layer as a navigable small world graph, which enables the number of hops scales logarithmically on the greedy search path. Similar to HNSW, HM-ANN constructs the graph with a fixed maximum number of links for each element, which guarantees that the average degree of each element in one layer is constant. The overall number of distance computation is proportional to a product of the number of hops and the average degree of the elements on the greedy path. Therefore, the search complexity in each layer of HM-ANN is logarithmic. Given a layer i with N_i elements, the search complexity of the layer i is $O(\log(N_i))$. Even with the bottom-up promotion, the maximum number of elements in each layer of HM-ANN remains N . Therefore, the overall search complexity of HM-ANN stays at $O(\log(N))$.

Index construction complexity. The construction of HM-ANN contains two passes over the dataset, due to the top-down insertions and the bottom-up promotions. The insertion of an element involves a graph traversal followed by a constant cost of inserting short-range and long-range links. Therefore, this phase has a cost of $O(N \log(N))$. The second pass of HM-ANN involves

degree calculation and ranking and then extracts elements with high-degree in L0 into upper layers. Calculating the degree of all elements and sorting them in terms of the degree at L0 is bounded by $O(N \times M + N \log(N))$. Therefore, in total the construction complexity of HM-ANN is $O(N \times M + N \log(N))$.

Memory usage complexity. HM-ANN stores connection and elements separately in slow and fast memories. In particular, HM-ANN stores the connections in L0 and the elements that only appear in L0 into slow memory, and stores connections and elements at upper layers into fast memory. The fast memory consumption of HM-ANN equals to the sum of memory consumption of each layer (except L0): $fast_memory_size = \sum_{i=1}^l (N_i \times M_i) \times byte_per_link + N_1 \times byte_per_element$, where N_i is the number of elements in layer i ($i > 0$), and M_i is the number of maximum established connections for each element in the layer i . The slow memory stores most of L0, which equals to $slow_memory_size = (N_0 \times M_0) \times byte_per_link + (N_0 - N_1) \times byte_per_element$.

3.4 Evaluation

3.4.1 Methodology

Testing bed. All experiments are done on a machine with Intel Xeon Gold 6252 CPU@2.3GHz. It uses DDR4 (96GB) as fast memory and Optane DC PMM (1.5TB) as slow memory.

Workloads. We use five datasets, BIGANN [90], DEEP1B [23], SIFT1M [90], DEEP1M [23], and GIST1M [18]. BIGANN contains one billion of 128-dimensional SIFT descriptors as a base set and 10,000 query vectors. DEEP1B contains one billion of 96-dimensional feature vectors of natural images and 10,000 queries. SIFT1M and DEEP1M are one-million subset vectors in BIGANN and DEEP1B respectively. GIST1M contains one-million 960-dimensional image descriptors.

Evaluation metrics. We measure the query response time as the average time of per-query execution time. We measure the accuracy with top-K recall (e.g., K=1, or 100), which measures the fraction of the top-K retrieved by the ANNS that are exact nearest neighbors.

Comparison configurations. For billion-scale tests, we include the following schemes: two state-of-the-art billion-scale quantization-based methods (IMI+OPQ [50] and L&C [51]); and the state-of-the-art non-compression-based methods (HNSW [128] and NSG [58]). To the best of our knowledge, directly running HNSW and NSG at billion-scale points would trigger the out-of-memory error, and no prior work has been able to run HNSW and NSG with the two billion-scale datasets on a single machine, without compression. We therefore create two baseline

configurations for both HNSW and NSG, using existing system-level data placement solutions: a *first-touch NUMA* configuration that places data in fast memory first until it is full and then in slow memory, and a *Memory Mode* configuration that treats fast memory as a hardware-managed fully-associative cache of slow memory. We include comparisons of HM-ANN at million-scale datasets with HNSW [128] and NSG [58], which are known to be the best-in-class solution on the three million-scale datasets.

3.4.2 Experiment Results

Billion-scale algorithm comparison. We compare HM-ANN with the graph- (HNSW and NSG) and quantization-based algorithms (IMI+OPQ and L&C). For HNSW, we build graphs with *efConstruction* and M set to 200 and 48 respectively; For NSG we first build a 100-NN graph using Faiss [4] and then build NSG graphs with $R = 128$, $L = 70$ and $C = 500$. We collect results on NSG and HNSW using Memory Mode, since it leads to overall better performance than using first-touch NUMA (see Section 3.4.3 for the comparison of the two). For IMI+OPQ, we build indexes with 64- and 80-byte code-books on BIGANN and DEEP1B respectively. We present the best search result with search parameters $nprobe=128$ and $ht=30$ for BIGANN and with autotuning parameter sweep on DEEP1B. For L&C, we use 6 as the number of links on the base level, and use 36- and 144-byte OPQ code-books. We use the same parameters (*efConstruction*=200 and $M=48$) as HNSW to construct HM-ANN. We set $efSearch_{L0}=2$ and vary $efSearch_{L1}$ to show the latency-vs-recall trade-offs.

Figures 3.1 (a)-(d) visualize the results. Overall, HM-ANN provides the best latency-vs-recall performance. Figure 3.1 (a) and (b) show that HM-ANN achieves the top-1 recall of $> 95\%$ within 1ms, which is 2x and 5.8x faster than HNSW and NSG to achieve the same recall target respectively. IMI+OPQ and L&C cannot reach the similar recall target, because of precision loss from quantization. As another point of reference, the SSD-based solution, DiskANN [198] (not open-sourced), provides 95% top-1 recall in 3.5ms. In contrast, HM-ANN provides the same recall in less than 1ms, which is at least $3.5\times$ faster. We compare top-100 recall shown in Figures 3.1 (c) and (d). HM-ANN provides higher performance than all other approaches. For example, it obtains top-100 recall of $> 90\%$ within 4 ms, while performs 2.8x and 5x faster than HNSW and NSG with the same recall target respectively. Quantization-based algorithms perform poorly and have difficulties to reach a top-100 recall of 30%.

Table 3.1 shows the index construction time and index size of HNSW, NSG, and HM-ANN. Among the three, HNSW takes the shortest time to build the graph. HM-ANN takes 8% longer

Table 3.1: Indexing time and memory consumption for graph-based methods on billion-scale datasets

	BigANN					DEEP1B				
	Indexing			Search		Indexing			Search	
	Graph size	Indexing time	Promo. rate	Fast-mem usage	Slow-mem usage	Graph size	Indexing time	Promo. rate	Fast-mem usage	Slow-mem usage
HNSW	475GB	90h	0.02	96GB (hw caching)	490GB	723GB	108h	0.02	96GB (hw caching)	748GB
NSG	285GB	115h	-	96GB (hw caching)	303GB	580GB	134h	-	96GB (hw caching)	599GB
HM-ANN	536GB	96h	0.16	96GB	462GB	756GB	117h	0.11	96GB	681GB

time than HNSW, because it takes an additional pass for the bottom-up promotion. However, HM-ANN is still faster to construct than NSG. In terms of memory usage, HM-ANN indexes are 5–13% larger than HSNW, because it promotes more nodes from L0 to L1. In terms of memory usage, HM-ANN consumes less fast memory than HNSW and NSG, which is valuable to reduce production cost [131, 183]. HNSW and NSG use all fast memory because they do not explicitly manage HM and by default using Memory Mode consumes all fast memory. The sum of slow and fast memory consumption can be larger than the index size, because there are metadata needed for search that are not counted into the index size.

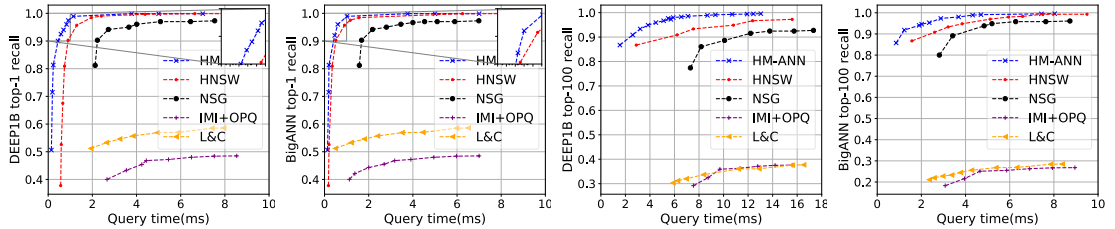


Figure 3.1: Query time vs. recall curve in (a) DEEP1B top-1, (b) BigANN top-1, (c) DEEP1B top-100, (d) BigANN top-100, respectively.

Million-scale algorithm comparison. Besides the billion-scale tests, we evaluate HNSW, NSG and HM-ANN with the three million-scale datasets, which can fit in DRAM. For HNSW and HM-ANN, we set $efConstruction$ and M to 100 and 16 for SIFT1M and DEEP1M; We set $efConstruction$ and M to 100 and 32 for GIST1M. For NSG we use parameters in [7] suggested by the authors to build the graph. Figure 3.2 shows the result. Overall, HM-ANN achieves competitive and sometimes even better performance as HNSW and outperforms NSG on all three million-scale datasets. We further verify that the total number of distance computation from HM-ANN is lower (on average 850/query) than that of HNSW (on average 900/query) to achieve 99% recall target. This indicates that HM-ANN provides better accuracy-vs-latency results even when the datasets can fit in DRAM.

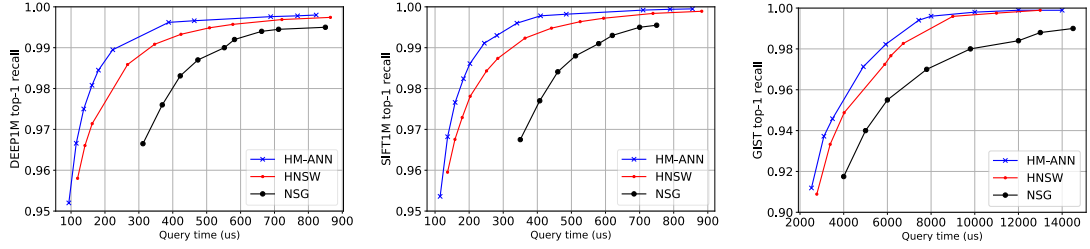


Figure 3.2: Query time vs. recall curve with (a)DEEP1M, (b)SIFT1M, and (c)GIST respectively.

3.4.3 Ablation Studies

Effectiveness of high-degree promotion. We compare the random promotion and high-degree promotion strategies. In this study, both strategies use the same number of promoted nodes for indexing and the same configurations for search. Figure 3.3 shows the results and indicates that high-degree promotion outperforms the baseline HNSW largely. The high-degree promotion performs 1.8x, 4.3x and 3.9x faster than the random promotion to reach 95%, 99%, and 99.5% recall targets, respectively, indicating that promoting high-degree nodes is effective for improving search efficiency.

T_{fast_mem} and T_{slow_mem} are measured by performing 10k distance computation in fast and slow memories and then report the average. T_{slow_mem} and T_{fast_mem} are 421ns and 183ns respectively.

HNSW with Parallel L0 search. We investigate whether it is sufficient to just modify the search procedure without modifying the hierarchical NN graph of HNSW to achieve similar performance gains as HN-ANN. Figure 3.4 shows the latency-vs-recall performance of default HNSW using parallel L0 search. We use T nearest neighbours found during HNSW L1 search as entry points for the parallel search in L0, where T is the number of parallel threads. We set $T = 4$, same as HM-ANN. HNSW with parallel L0 search only slightly outperforms HNSW. This suggests that parallel L0 search alone is not sufficient for performance improvement. Without it, the elements of L1 in HNSW are selected randomly and sparse, and the entry nodes found through L1 search are sub-optimal. As a result, even though the parallel search in L0 searches more nodes under the same time, the accuracy only slightly improves.

Performance benefit of memory management techniques in HM-ANN. Figure 3.5 contains a series of "stepping stones" between HNSW and HM-ANN to show how each optimization of HM-ANN contributes to its improvements. "HNSW + Bottom-up promotion (BP)" modifies the HNSW algorithm, mapping the bottom-most layer (i.e., L0) to the slow memory while building a high-quality projection of L0 in fast memory without significantly impacting search efficiency. It

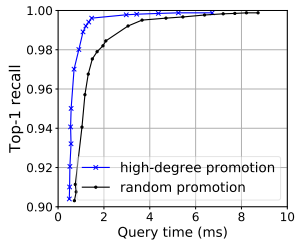


Figure 3.3: Comparison of two promotion strategies.

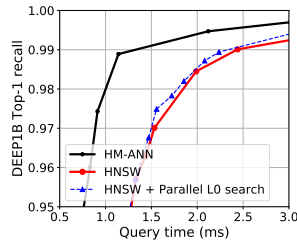


Figure 3.4: HNSW with parallel L0 search.

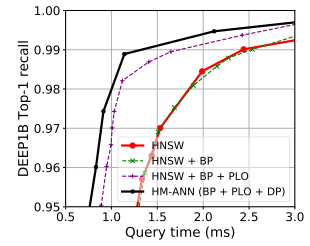


Figure 3.5: Comparison of techniques in HM-ANN.

provides the benefit of improved search quality in fast memory while providing better entry points to L0 search in slow memory. Together with the parallel L0 search (i.e., “HNSW + Bottom-up promotion (BP) + Parallel L0 search (PLO)”) it significantly improves the search efficiency versus running HNSW on HM without explicit data management. For example, to reach a 99% recall target, HM-ANN reduces the query time by 1.75x compared with HNSW. Finally, by prefetching data from slow memory to fast memory, HM-ANN further pushes the search efficiency frontier.

System level data management solutions. We compare HM-ANN with HNSW in Memory Mode and first-touch NUMA (as a software-based solution to manage data placement in HM, HM-ANN does not work with Memory Mode and first-touch NUMA). We also evaluate HNSW on slow memory without using any DRAM. Figure 3.6 shows the result. The figure shows that HM-ANN outperforms HNSW with Memory Mode and first-touch NUMA by 2x and 3.7x while achieving top-1 recall above 95%. The results suggest that although HM enables large memory capacity, simply using a system-level solution without algorithm change cannot make the best use of HM. Explicitly managing data for HM as HM-ANN does is the key to achieve superior latency and recall results.

Effectiveness of performance model-guided search. Figure 3.7 shows the distribution of ($efSearch_{l_0}$, $efSearch_{l_1}$) pairs that meet time constraint of $<1\text{ms}$ and recall constraint of $\geq 90\%$. The bottom-left and top-right regions include those pairs violating either recall or time constraint; The colored regions are those meeting the constraints; The darker color has shorter query time. Figure 3.7 shows the performance model removes most of configurations violating the constraints.

To show effectiveness of performance modeling, we evaluate HM-ANN with BIGANN and 5 latency constraints from 1ms to 5ms (vertical red lines) in Figure 3.8. Red triangles represent ($efSearch_{l_0}$, $efSearch_{l_1}$) that meet the latency constraints set by Eqn. 3.1. Among those, we list 9 recall constraints marked with horizontal blue lines. For those recall constraints, 9 five-stars

are those selected by HM-ANN, which meet the corresponding recall constraints while also having the shortest query time.

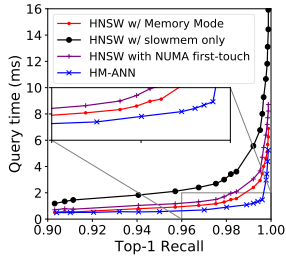


Figure 3.6: Comparison of data management methods.

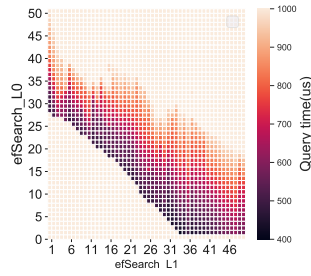


Figure 3.7: Distribution of $efSearch$.

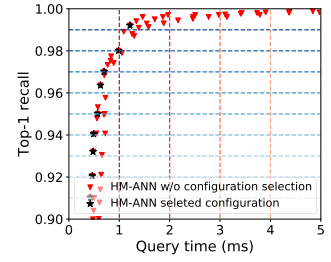


Figure 3.8: Performance with various $efSearch$.

3.5 Conclusions

HM can store billions of point database in a single machine. However, indexing and search algorithms on HM must be re-designed to release large performance potential of HM. We present a new graph-based indexing and search algorithm called HM-ANN, which maps the hierarchical design of the graph-based ANNs with memory heterogeneity in HM. Furthermore, HM-ANN adjusts the amount of distance computations at different layers to allow most accesses happen in upper layers stored in fast memory. Combined with a set of system-level techniques, HM-ANN is able to avoid expensive accesses in slow memory without sacrificing accuracy. Evaluation on billion-scale datasets show that HM-ANN establishes the new state-of-the-art for indexing and searching billion point datasets.

Chapter 4

Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning

Memory capacity is a major bottleneck for training deep neural networks (DNN). Heterogeneous memory (HM) combining fast and slow memories provides a promising direction to increase memory capacity. However, HM imposes challenges on tensor migration and allocation for high performance DNN training. Prior work heavily relies on DNN domain knowledge, unnecessarily causes tensor migration due to page-level false sharing, and wastes fast memory space. We present Sentinel, a software runtime system that automatically optimizes tensor management on HM. Sentinel uses dynamic profiling, and coordinates operating system (OS) and runtime-level profiling to bridge the semantic gap between OS and applications, which enables tensor-level profiling. This profiling enables co-allocating tensors with similar lifetime and memory access frequency into the same pages. Such fine-grained profiling and tensor collocation avoids unnecessary data movement, improves tensor movement efficiency, and enables larger batch training because of saving in fast memory space. Sentinel reduces fast memory consumption by 80% while retaining comparable performance to fast memory-only system; Sentinel consistently outperforms a state-of-the-art solution on CPU by 37% and two state-of-the-art solutions on GPU by 2x and 21% respectively in training throughput.

4.1 Introduction

Deep neural networks (DNN) have been shown preliminary success in many fields. However, training those models can be extremely memory-consuming. For example, the recent language models and translation models have 100s of billions of parameters [193] requiring 100s of GB of memory for training. Although it has been repeatedly demonstrated that larger models and more data lead to improved model accuracy on many tasks [43, 74, 170], the memory becomes a major bottleneck either when training models with more weight parameters or with larger batch sizes. Lack of memory causes DNN training to have out-of-memory crashes and limits the sizes of the model and batch for training, causing degradation in training effectiveness and efficiency [182, 218]. Adding more DRAM can mitigate the problem, but often comes with huge costs. In this work, we look into overcoming the memory scaling issue for DNN training by leveraging heterogeneous memory (HM) to achieve larger memory capacity.

HM is an emerging memory architecture. Within HM, multiple memory components with different technologies are combined to construct main memory. HM is typically composed of a high-capacity memory (but with relatively worse performance, such as non-volatile memory) and a high-performance memory (but with smaller capacity, such as DRAM). HM brings a promising solution to increase memory capacity and avoids the limitation of existing memory technologies.

HM for DNN training has been explored by several studies [77, 79, 106, 133, 158, 182, 218]. Most of them focus on mitigating GPU-side memory space limitation by leveraging larger CPU-side system memory [79, 106, 133, 158, 182, 218], while a recent study demonstrates a HM that uses a persistent memory to scale the CPU-side DRAM capacity [77]. As observed in these studies, computation efficiency of using HM requires a careful memory management, such as timely tensor placement and migration, subject to the access patterns of tensors and the performance disparity of different memory components. More specifically, existing solutions explore methods to proactively release and prefetch temporarily inactive tensors, determine inactive tensors based on DNN topology, and find data swap time between memories by analyzing tensor access order [77, 79, 106, 158] or using detailed domain knowledge [133, 182, 218].

However, there has not been a study that thoroughly evaluates individual tensor characteristics and the semantic gap between operating system/architecture and memory management in deep learning frameworks. Missing this study leaves many performance improvement opportunities on the table. For example, most of the existing solutions focus on the order of tensor accesses to determine the timing and target tensors to swap. However, such solutions are oblivious to tensor characteristics such as number of tensor accesses in memory and tensor lifetime, which would

lead to unnecessary tensor migrations. For example, some tensors such as short-lived ones might be better to stay in fast memory rather than unnecessarily migrated to be used only a few times or never used again.

More importantly, most of the existing studies tackle tensor placement at the granularity of individual tensors. However, as the memory management of operating system (OS) and underneath architecture is conducted at the page level, such a tensor-level mapping would lead to memory fragmentation or unexpected tensor migration if multiple tensors having different access patterns are mapped to the same page. For example, if there are four pages, where 25% of each page is mapped with tensors frequently accessed in the similar time and having similar lifetime, all four pages would need to be placed in fast memory, even though the remainder of the four pages is filled with rarely accessed tensors whose lifetimes are different from that of the frequently accessed tensors. However, if we co-locate all frequently accessed tensors to one page in this example, placing this single page in fast memory would be sufficient for high performance training.

To address the aforementioned issues, we propose *Sentinel*, a software runtime system that automatically manages and optimizes tensor migration and allocation in HM, and allows to train DNN models with a much smaller size of fast memory but achieves performance similar to that on the fast memory-only system. To do that, we first conduct an extensive study on *workload characteristics of DNN*. We observe that there are a large number of small (less than one page) and short-lived (lifetime shorter than one layer) tensors. On the other hand, the peak memory consumption of frequently accessed tensors (hot tensors) is not big (tens of MB), and tensors commonly share pages but with different access frequencies. These observations indicate that an ideal memory management strategy for DNN training should co-locate tensors with similar lifetime and access frequency in fast memory while minimizing page-level false sharing to reduce peak consumption of fast memory. To our best knowledge, this is the first in-depth tensor and memory mapping characterization of DNN training; It is generally applicable on linear and non-linear network topologies and includes *all* tensors, which is different from those of existing studies [38, 92, 182, 218] that focus on specific tensors (e.g., input tensors of convolution operations or model weights) on certain DNN.

Driven by the observations, Sentinel enables efficient DNN training with three major innovations. First, Sentinel implements a *tensor-level dynamic profiling* to collect characteristics of individual tensors which is impossible in the traditional page-level profiling. This method bridges the semantic gap between OS and DNN application. It allows the runtime to associate tensors with

the DNN topology, dynamically identifying long-lived but sparsely accessed tensors that can be migrated. More importantly, our profiling method counts tensor accesses in memory (i.e., the number of accesses to each tensor in memory), not just checks whether tensors are referenced in operations as in many HM management solutions for DNN training [77, 79, 92, 158, 182, 218, 244]. Counting tensor accesses in memory is fundamental for memory optimization for DNN, because it leads to new optimization techniques, such as tensors co-location and being graph-agnostic.

Second, Sentinel improves *tensor migration efficiency by avoiding page-level false sharing* and unnecessary tensor movement, which is often ignored in related work [77, 79, 92, 158, 182, 218, 244]. Sentinel aggregates small tensors having a similar lifetime and access count into the same page to prevent page-level false sharing. Sentinel also pins short-lived tensors to a reserved fast memory space to prevent their unnecessary movement to slow memory. Note that the unnecessary movement of short-lived tensors are commonly observed in existing page-level data migration [234] and hardware-managed caching mechanisms [83, 175, 238], which leads to memory bandwidth waste and performance loss.

Third, Sentinel employs a *performance model-directed proactive migration strategy*. Similar to existing solutions [158], Sentinel dynamically moves unused tensors out of fast memory and moves to-be-used tensors into fast memory to save its space. However, unlike existing studies [182, 218], we consider performance trade-off between migration frequency and performance benefit, as frequent tensor movement can be exposed to the critical path and cause performance loss. To identify the optimum migration interval that not only reduces memory capacity but also avoids performance loss, we introduce analytical performance models that allow exploration of various migration intervals and effectively find the optimum one with negligible runtime overhead. The performance models bring great flexibility and high performance for tensor migration across layers for various DNN topologies. The tensor migration is controlled purely subject to the performance models to maximize overlap between tensor migration and DNN training, unlike existing studies that use migration algorithms heuristically designed for given network topologies [38, 92, 182, 218] or limited memory capacity [79, 158].

In summary, the key contributions are as follows.

- **Characterization study.** We systematically analyze how tensors are allocated and accessed in TensorFlow.
- **Runtime system.** We introduce a runtime system, Sentinel, which is featured with a novel profiling method counting memory accesses at the tensor level. Guided by analytical performance

models, Sentinel enables efficient tensor migration by avoiding page-level false sharing and unnecessary data movement.

- Evaluation.** We evaluate Sentinel on two HM systems: one is based on DDR4 (fast) and Optane DC persistent memory (slow), and the other is based on NVIDIA V100 GPU (fast) and CPU (slow). On the Optane-based system, we show that using only 20% of peak memory consumption of DNN models as fast memory size (a 5X reduction), Sentinel achieves similar performance (9% performance difference on average) to DRAM-only system. Furthermore, Sentinel consistently outperforms two state-of-the-art solutions (IAL [234] and AutoTM [77]) as well as DRAM-cached Optane (using DRAM as a hardware cache) and first-touch NUMA policy, by 37%, 17%, 23% and 70% in training throughput, respectively. On the GPU-based system, Sentinel enables larger batch size in training by 1.9X and higher training throughput by 2X than vDNN [182], and enables comparable batch size and higher training throughput by 16%, 17% and 65% than three state-of-the-art solutions (Capuchin [158], AutoTM [77] and SwapAdvisor [79]), respectively.

4.2 Background

Training DNN models. A typical DNN model comprises of a stack of *layers*, each of which is a group of neurons. Each neuron in a layer computes a non-linear function of the outputs of neurons in the preceding layer, using a set of weights. Training DNN often involves a large number of training iterations (each iteration is a training step). In each step, a *batch* of training samples are fed into DNN. Performance of each step (e.g., execution time and memory access pattern) remains stable across steps, hence highly predictable [120, 122, 197]. Training DNN often uses a framework, such as TensorFlow [11] and PyTorch [64]. These frameworks use a dataflow execution model where the workload of DNN is modeled as a directed graph. *Operations*, such as 2D convolution, matrix multiplication, and array concatenation, are implemented as primitives. Those operations are represented as nodes in the graph. Within the graph, edges between nodes capture dependencies between nodes.

Recent efforts. Heterogeneous memory is used for DNN training recently [77, 79, 92, 158, 182, 218]. We comprehensively compare state of the art with Sentinel in Table 4.1 from multiple perspectives. In Table 4.1, dynamic profiling captures the effects of inter-operation parallelism on memory accesses and is generally applicable on various input data sizes and architectures, which are often missed in static profiling. Minimization of fast memory usage means making best efforts

Table 4.1: Comparison between existing work on HM management for DNN training.

	Dynamic profiling	Fast memory usage minimization	Graph agnostic	Count tensor accesses in memory	Page-level false sharing avoidance
vDNN [182]	N	N	N	N	N
Superneurons [218]	N	N	N	N	N
Layrub [92]	N	N	N	N	N
SwapAdvisor [79]	N	N	Y	N	N
AutoSwap [244]	N	N	Y	N	N
AutoTM [77]	N	Y	Y	N	N
Capuchin [158]	Y	N	Y	N	N
HALO [69]	Y	N	Y	N	N
Sentinel	Y	Y	Y	Y	Y

to reduce fast memory size; This also means targeting on all tensors (not just a few tensors such as feature maps) to look for migration opportunities. Being graph agnostic means there is no need of detailed DNN knowledge (such as which tensor is feature map or weight), which makes the solution more general, instead of just for some specific DNN models. Counting tensor accesses in memory totals the number of memory accesses at data object level, which is much more than just checking whether data objects are referenced in operations [69, 77, 79, 158, 182, 218]. Counting tensor accesses provides optimization opportunities to co-locate tensors and prioritize data migration. Avoiding page-level false sharing is necessary to improve page migration efficiency and achieve additional savings of fast memory usage, revealed in Section 4.3.2. Sentinel excels, because it uses dynamic profiling, count tensor accesses in memory, and is graph agnostic; Sentinel has high migration efficiency and enables larger model (or larger batch) training. Sentinel is applicable to both CPU and GPU, as demonstrated in this paper, while some state-of-the-arts (e.g., Capuchin [158]) focus only on GPU and use expensive recomputation to save GPU memory, whose effectiveness on CPU remains to be studied.

There are large differences between generic memory management (GMM) (e.g, tcmalloc [60] and garbage collection (GC) in a managed language/runtime) and Sentinel: (1) Tensor lifetime management in GMM lacks DNN semantics and hence misses opportunities to timely migrate tensors to avoid performance loss or save fast memory, hence fails to minimize fast memory usage, evidenced in Table IV; (2) GMM cannot work well on GPU at tensor levels; (3) Without coordination of OS, GC ignores the impact of CPU cache hierarchy on main memory accesses; (4) Current DNN training frameworks are not based on managed runtime and cannot easily employ GC.

4.3 Analysis and Characterization of Main Memory Accesses in DNN

We characterize main memory accesses to drive our design.

4.3.1 Profiling Framework

We build a profiling framework. It is integrated into TensorFlow and used in a profiling phase (one training step) to direct tensor management at runtime (Sec. 4.4). The profiling framework collects the following information: (1) the number of *main memory accesses* per tensor, (2) tensor size and (3) lifetime. To collect the above information, the profiling framework includes the support at *both* OS and TensorFlow runtime levels. At OS level, Sentinel collects the number of memory accesses at the page level. This is implemented by a software-only solution. In particular, to track a page for access counting, Sentinel sets a reserved bit (bit 51) in its PTE (i.e., poisoning PTE) and then flush the PTE from TLB. When the page is accessed, a TLB miss occurs and triggers a protection fault. Sentinel uses a customized fault handler to count this page access, poisons the PTE, and flushes it from TLB again to track next page access.

To bridge the semantic gap between OS and DNN framework, each memory page has only one tensor (but a tensor can use more than one pages). This is implemented by making object allocation aligned with memory page. Using this method, page-level profiling becomes tensor-level profiling. This method slightly increases memory footprint (Sec. 4.7) but it only happens during the profiling phase of Sentinel on slow memory. After the profiling phase, tensors are re-organized to reduce memory footprint and improve performance. Data reorganization happens *during memory allocation* (Sec. 4.3.2), and hence does not stop training process and does not impact performance. The profiling method does not increase the consumption of fast memory.

At the TensorFlow runtime, Sentinel leverages memory (de)allocation to get the size and lifetime of tensors. Moreover, Sentinel introduces an API that allows the user to annotate DNN to indicate the end of each layer in DNN. Based on the above infrastructure, Sentinel is able to associate a tensor with the DNN topology (i.e., we can know which layer(s) a tensor is alive), which is helpful to direct tensor migration.

Our profiling method uses only one training step for profiling. During the profiling, Sentinel captures each page read and write by repeatedly poisoning the page. This is expensive because of system calls and TLB misses. However, it does not lose profiling accuracy. Also, considering that a typical DNN training involves millions of training steps, the profiling overhead is easily

amortized. The traditional profiling methods face a dilemma between profiling overhead and accuracy. In particular, frequently collecting memory access information brings high profiling accuracy at the cost of large runtime overhead, and vice versa [14, 78, 226, 227]. Leveraging the repetitiveness of DNN training, Sentinel breaks the dilemma, and enables both high profiling accuracy and low profiling overhead.

Our profiling method is featured with the coordination between OS and TensorFlow runtime. This provides accurate profiling, which is unachievable by TensorFlow runtime alone. In particular, OS allows us to track memory accesses filtered by processor caches; Working with the coordination between OS and TensorFlow runtime, we do not need to handle pointer aliasing commonly found in TensorFlow implementation, which is difficult to be handled by a runtime solution.

4.3.2 Observations and Preliminary Analysis

We profile DNN models listed in Table 4.3 and have the following observations to guide our design:

Observation 1: There are a large number of small tensors with short lifetime in DNN training workloads.

We define a tensor as small if it is smaller than a page size. A tensor is alive after it is allocated and before it is freed. We define the lifetime of a tensor in terms of the number of layers where the tensor is alive. In the rest of the paper, *we define short-lived tensor as those whose lifetime is no longer than one layer*. Taking ResNet-32 as an example (its configuration is in Table 4.3), 92% of its tensors have lifetime no longer than one layer. Among them, 98% is small tensors. The peak memory consumption of short-lived tensors is small, and typically bounded by a few GB.

Observation 2: The uneven distribution of hot and cold tensors provides opportunities for tensor management.

For example, 52.3% of tensors (using 907 MB, which is 54% of total memory pages) in ResNet-32 are accessed less than 10 times in main memory. On the other hand, some tensors in ResNet-32 are frequently accessed (having > 100 accesses), taking only 4 MB (0.2% of total memory pages). They are the candidates to be placed into fast memory, and their size is a small portion of total memory pages.

Observation 3: Page-level false sharing exists in DNN. The page-level profiling (not tensor-level) for tensor management can be misleading.

For example, in ResNet-32, if we perform tensor-level profiling, in a training step, for those

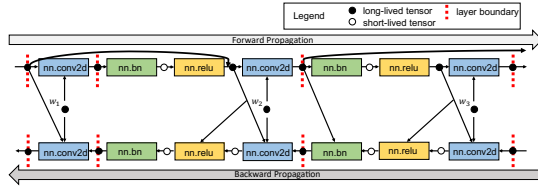


Figure 4.1: An example to show tensor access patterns across layers in ResNet-32. This figure shows the first three layers and the last three layers in ResNet-32.

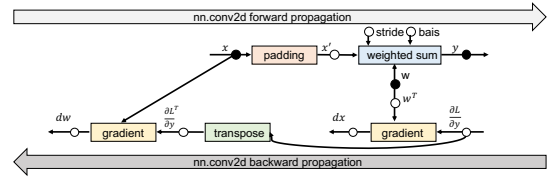


Figure 4.2: Data processing in a TensorFlow operation, `nn.conv2d`.

less-frequently accessed tensors that have only 1-10 accesses in main memory, their total object size is 908 MB. However, if we perform page-level profiling, in the same training step, for those memory pages with 1-10 accesses in main memory, their total page size is 764 MB. This indicates that some less-frequently accessed tensors fall into some pages that are counted as more frequently accessed in page-level profiling. Hence, if one uses page-level profiling to guide data management, those less-frequently accessed tensors can be placed into fast memory and waste memory space and bandwidth. We refer to the above result as page-level false sharing.

Example. We use ResNet-32 as an example to characterize tensors. Figure 4.1 shows operations in six layers; Figure 4.2 shows tensor processing in Operation `nn.conv2d` commonly used in DNN’s forward and backward propagation.

Short-lived tensors. We find two cases. (1) Inside an operation, tensor processing (e.g., padding and transpose shown in Figure 4.2, expansion, concatenation and squeeze) often generates short-lived tensors, which are only used in that operation; (2) The output tensor of some operation is short-lived, exemplified by the output of batch normalization (i.e., `nn.bn` in Figure 4.1). Memory allocation and free for a short-lived tensor always happen in one layer.

Long-lived tensors. We find two cases. (1) Weights associated with each layer (shown as “w1” and “w2” etc. in Figure 4.1). They are allocated before training steps, and updated throughout them. (2) Intermediate results generated in a layer and consumed by the downstream operations in another layers. An example is the output tensors of operations `nn.conv2d` and `nn.relu` in the forward propagation layers shown in Figure 4.2. These output tensors are consumed by the backward propagation layers to calculate gradients. The memory space for these intermediate results is allocated when they are generated and then freed after they are consumed.

Memory access patterns. Memory accesses to tensors are associated with layers. Memory accesses to short-lived tensors tends to be *ephemeral* and *bursty*, which means in a layer, there can be a number of short-lived tensors created, accessed a few times, and freed. Memory accesses

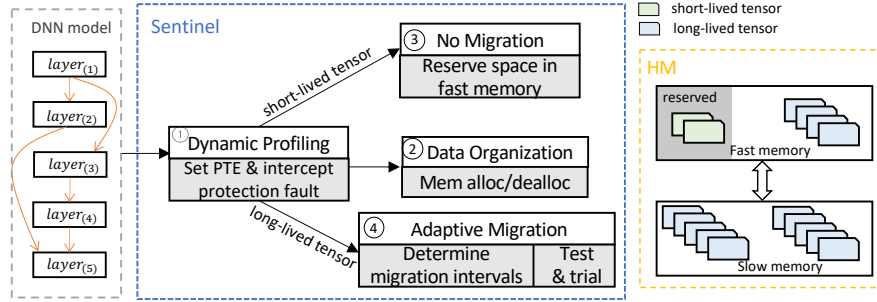


Figure 4.3: Overview of Sentinel. The white and shadow boxes represent functionality and mechanisms, respectively.

to a long-lived tensor tend to be *sparse* and *periodical*, which means memory accesses happen in a couple of specific layers, but not all layers. In addition, memory allocations for long-lived intermediate results and short-lived tensors are interleaved throughout the training process, which causes page-level false sharing.

Design choices. Profiling results motivate us to make three design choices. (1) We choose DNN layer as the basic granularity for tensor management, given the fact that lifetime and memory access patterns of tensors are associated with layers. This choice brings convenience for tensor prefetching and migration overhead controlling. (2) We treat tensors differently, instead of using a unified policy to manage them as in [14, 78, 95, 226, 227, 234]. This choice allows us to enable high performance and minimize fast memory capacity. (3) We do not use static analysis as in [216] to decide data placement, because static analysis lacks timing information needed to overlap data migration and computation; It cannot accurately capture main memory accesses and ignores the impact of thread-level parallelism on data locality.

4.4 Design

4.4.1 Overview

Figure 4.3 overviews Sentinel. Sentinel uses dynamic profiling (Sec. 4.4.2) to collect the number of main memory accesses at tensor level and lifetime of tensors based on customized memory allocation. The dynamic profiling uses one training step to collect the information. After that, Sentinel re-organizes memory allocation for short-lived tensors to facilitate tensor management and avoid page-level false sharing.

Driven by the profiling results, Sentinel treats short- and long-lived tensors separately. Short-

lived tensors are allocated in contiguous memory space in fast memory and not involved in tensor movement. This method (Sec. 4.4.3) avoids unnecessary tensor movement. To handle long-lived tensors, Sentinel uses an adaptive migration algorithm (Sec. 4.4.4). It partitions each training step into many migration intervals based on DNN model topology. In a migration interval, Sentinel migrates tensors needed for the next interval, overlapping application execution with tensor migration. Sentinel must determine an appropriate migration interval length¹, such that tensors can be timely migrated from slow to fast memory before they are needed by the next migration interval and without running out of fast memory. We formulate the problem and determine the optimum length. We also use a test-and-trial algorithm to determine if the migration cannot finish before the next interval, whether continuing migration can lead to better performance.

4.4.2 Dynamic Profiling and Data Reorganization

Sentinel integrates the profiling framework into TensorFlow. Based on the profiling results, Sentinel uses a customized memory allocation policy in the remaining training steps, described as follows. (1) For those short-lived tensors alive in the same layer, they are allocated into the same pages, because of their similarity in lifetime and the number of memory accesses. (2) For those long-lived tensors that reside in the exactly same layers, we use the following algorithm to determine their memory co-allocation. We first sort them in terms of the number of memory accesses in descending order, and then allocate them in contiguous memory pages, following the order. As a result, tensors with the similar memory access pattern can be allocated into the same memory pages. (3) For those long-lived tensors that do not reside in the same layers, they never share any memory page. (4) Long- and short-lived tensors never share any memory page.

The above data reorganization happens to long- and short-lived tensors allocated in the middle of the training. Those tensors are allocated and freed in each training step, allowing Sentinel to reorganize them across training steps without impacting program correctness. A few long-lived tensors (e.g., weights and input samples) are allocated before the training process; They cannot be reorganized in the middle of training, because that changes memory addresses of the tensors and causes wild pointers. Sentinel ensures that these tensors never share pages to avoid page-level false sharing.

¹We distinguish *migration interval* and *migration interval length* in the rest of the paper. The number of layers in a migration interval is the migration interval length.

4.4.3 Handling Short-Lived Tensors

During training, an individual short-lived tensor is not accessed many times (e.g., less than 10 times in ResNet-32) in main memory, compared to many long-lived tensors. Our profiling results show that there are a large amount of short-lived tensors throughout the whole training, and they share the same memory access characteristics (i.e., short life time, small size, and infrequent accesses in main memory). We must use a general policy to manage them.

We use the following algorithm to manage short-lived tensors. We allocate a continuous memory space in fast memory for them. Tensors in this space are never considered for migration. This space is reused for short-lived tensors as they are allocated and freed throughout the training. The space is reserved at the beginning of each migration interval to accommodate short-lived tensor in the interval. Doing this, Sentinel guarantees that there is always memory space for short-lived tensors (i.e., no competition from long-lived tensors), because the placement of short-lived tensors is critical for performance. Within a migration interval, the space can be dynamically shrunk to free space for long-lived tensors, when a memory page in the space is no longer needed by short-lived tensors.

The above method addresses the limitation of the existing methods that use a caching algorithm [78, 95, 175, 246] to decide tensor placement. Those methods move short-lived tensors to slow memory, even though they are not accessed any more. This causes unnecessary tensor movement and wastes memory bandwidth. Furthermore, short-lived tensors unnecessarily stay longer in fast memory, wasting valuable space in fast memory. The above problem is caused by the fact that making the decision on the movement of short-lived tensors takes some time, due to the necessity of counting memory accesses to run the caching algorithm. Also, counting memory accesses for individual tensors can be inaccurate, because tensors with different memory access patterns share memory pages.

In our design, fast memory is always large enough to host short-lived tensors. If not, short-lived tensors will be frequently moved between fast and slow memories. This tensor movement is highly inefficient in terms of both performance and energy efficiency. Hence, we assume that the fast memory size is at least larger than the peak memory consumption of those short-lived tensors (discussed in Section 4.4.5). Since short-lived tensors are frequently allocated and freed and we reuse the same memory space to host them, the size of peak memory space for short-lived tensors is small, and typically bounded by a few GBs, making it feasible to host them in fast memory without consuming too much space.

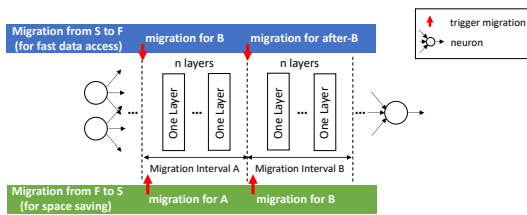


Figure 4.4: Tensor migration based on the migration intervals. “S” and “F” stand for slow and fast memories respectively.

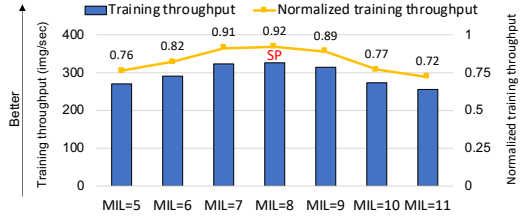


Figure 4.5: Performance (training throughput) variance as we change the migration interval length (MIL). “SP” stands for sweet spot (the optimum migration interval length).

4.4.4 Adaptive Layer-Based Migration

We migrate long-lived tensors because they are used sparsely and periodically. The tensor migration is controlled by the migration interval length which determines how frequently we migrate tensor between fast and slow memories. In particular, we partition a training step (i.e., one forward step plus one backward step) into equal-sized intervals, exemplified in Figure 4.4.

Tensor migration from slow to fast memory is triggered at the beginning of each interval, aiming to prefetching tensors needed by the next interval into fast memory before *the next interval* starts. Tensor migration follows a decreasing order of tensors in terms of number of memory accesses to each tensor, such that tensors with the largest number of memory accesses are migrated to fast memory first. The order information is available after data reorganization (Sec. 4.4.2). Following this order for migration allows Sentinel to make the best use of fast memory for high performance, in case certain tensors are left out in slow memory, which is discussed later. The tensor migration is overlapped with DNN training computation as much as possible, such that the overhead of tensor migration is removed from the critical path.

Tensor migration from fast to slow memory happens in the middle of the interval, when the long-lived tensor is no longer accessed by any operation in the interval. Such tensor migration is used to save fast memory space as much as possible, in order to accommodate upcoming tensor migration. We can know if a long-lived tensor will be used by any operation in an interval by using the profiling results.

We define the migration interval in terms of layers in DNN, not in terms of execution time, because of the following three reasons. First, the layer-based migration interval naturally guarantees the completion of operations at the end of the interval, because no operation runs across layers. The time-based migration interval cannot guarantee that, and hence needs inevitable

synchronization between application execution and tensor migration, causing performance loss. Second, each layer is associated with a computation phase with a memory access pattern (e.g., which tensors are accessed and their lifetime). The layer-based migration interval allows us to easily leverage the memory access patterns collected at the profiling phase to guide tensor migration. Third, the time-based migration imposes challenges on deciding which operations are executed in which migration interval, because of operation-level parallelism.

Determining an appropriate migration interval length is challenging. If the migration interval length is too long or too short, we cannot achieve the best performance. Figure 4.5 shows the performance when we use different interval lengths to train ResNet-32 on an Optane-based platform (shown in Table 4.2). There is a 21% performance variance when we change the interval length from 5 to 11. When the interval length is 8, we achieve the best performance. Hence, determining an appropriate interval length is critical for performance.

We analyze the trade-off between long and short migration interval lengths as follows. If the interval length is long, then the tensors to migrate for an interval is large. The interval length cannot be too long. Otherwise the tensors to migrate can be larger than the available space in fast memory. This constraint on the interval length is the *space constraint*, formulated in Equation 4.1.

If the interval length is short, then the available execution time to overlap tensor migration with application execution is short. The interval length cannot be too short. Otherwise, the tensor migration time is largely exposed to the critical path. We want to minimize the migration time exposed to the critical path, which is formulated in Equation 4.2.

In Equations 4.1 and 4.2, RS is the fast memory space for short-lived tensors, S is the fast memory size, and MIL stands for the migration interval length. RS is a function of the migration interval length (different migration interval lengths have different RS). In Equation 4.1, $Tensor$ is the size of tensors for migration in an interval; In Equation 4.2, BW is the migration bandwidth from slow to fast memory, and T is the DNN training time in an interval. $(S - RS(MIL))/BW$ is the tensors migration time. $Tensor$ and T are functions of the interval length (different interval lengths have different $Tensor$ and T).

$$\text{Space constraint: } Tensor(MIL) < S - RS(MIL) \quad (4.1)$$

$$\text{Goal: } \arg \min_{MIL} ((S - RS(MIL))/BW - T(MIL)) \quad (4.2)$$

RS is relatively stable, according to our profiling results: There is only a small variance as

we change MIL . Hence $S - RS(MIL)$ is near constant. $Tensor(MIL)$ and $T(MIL)$ are monotonically increasing functions of MIL (i.e., a larger MIL indicates larger $Tensor$ and T , and vice versa).

After collecting the profiling results, Sentinel uses Equation 4.1 to narrow down the search space of finding the optimum migration interval length; Sentinel then uses profiling results to estimate performance of using various interval lengths, based on which to determine the best one according to Equation 4.2. This exploration is quick, because it does not need to run any training operations, and the search space has only one dimension (i.e., the migration interval length). Due to the quick exploration and low-dimensional search space, using a statistical algorithm such as the genetic algorithm or Markov Chain Monte Carlo for multi-dimensional space as the existing work [79, 91] is not necessary.

We cannot use training steps to try every possible migration interval length to determine the best one without using performance modeling, because it raises concerns on runtime overhead, when the number of layers (and sub-layers that can be used as an interval) in a DNN model is very large.

We encounter three possible tensor migration cases at the end of a migration interval. We discuss them as follows. Assume that we have two intervals, A and B , and B is right after A . Sentinel migrates tensors at the beginning of A for B . At the end of A , we have three cases.

- Case 1: All tensor migration has been finished;
- Case 2: Tensor migration cannot finish, because of lack of space in fast memory;
- Case 3: Tensor migration cannot finish because of lack of time for migration (there is still space in fast memory).

In Case 1, once B starts, all of the migrated tensors are in fast memory, which is the ideal case. Case 2 can happen, even though the space constrain is respected for tensor migration for B , because some tensors in A may not be timely migrated from fast to slow memory to save space for B ; Case 3 can happen, because the optimization goal ensures the migration time exposed to the critical path is minimized but the migration may not necessarily finish when B starts. We must avoid Cases 2 and 3 for the best performance. The migration interval length has impact on how often the three cases happen. Given a fast memory size, a short interval length can create more Case 3 while a long interval length can create more Case 2.

To avoid Case 2, long-lived tensors are immediately moved out of fast memory in the middle of A to save space, once the remaining operations in A do not need them. This solution prevents

all occurrences of Case 2 in our evaluation.

To handle Case 3, we can either continue migration and let B wait for the migration completion, or leave tensors in slow memory. The continuation of tensor migration exposes tensor migration into critical path, but the execution of B uses tensors in fast memory; On the contrary, leaving tensors in slow memory uses the tensor in slow memory but avoids tensor migration overhead. This is a classic trade-off between data locality and data movement. To determine which method leads to the better performance, we use a test-and-trial algorithm.

In particular, whenever Case 3 happens at the end of an interval, we use one training step to try the continuation of tensor migration, and use another training step to try no-tensor-migration. We measure the performance of the two methods and use the better method in the remaining training steps.

The above algorithm does not cause significant runtime overhead, because Sentinel uses at most two training steps for test and trial to handle each occurrence of Case 3 and the number of occurrences is less than 10 in our evaluation, while the total number of training steps is easily millions. We quantify runtime overhead in Sections 4.7.2 and 4.7.3.

4.4.5 Discussions

The lower bound on fast memory size. Although fast memory can be smaller with Sentinel, there is a lower bound on fast memory size to avoid significant performance loss. This lower bound is the peak memory consumption of short-lived tensors among all migration intervals plus the largest long-lived tensor. Smaller than this bound, the runtime system has to either frequently migrate short-lived tensors or has no space to accommodate long-lived tensors, which easily causes performance loss larger than 20%.

Handling dynamic graphs. Sentinel focuses on common DNN models with static graphs, similar to other work [77, 197]. Some frameworks, such as PyTorch and TensorFlow 2.0, support dynamic graphs. Depending on the input size within a batch, these frameworks generate a different dataflow graph with a right shape to accommodate the batch. Hence there could be multiple graphs. To handle dynamic graphs, the existing solution pads zeros at the end of input [65], such that batches have the same structure. This transforms a dynamic graph into a static one, but at the cost of larger memory footprint and unnecessary computation. We use a solution similar to [197] that uses bucketed profiling. In particular, Sentinel bucketizes input sizes into a small number of buckets (at most 10). Input sizes in the same bucket have a similar graph. Sentinel profiles each bucket to decide tensor migration.

Handling control dependencies. A static graph can have control flow. Depending on input values in a batch, the graph can have different dataflow, causing different memory access patterns. Sentinel handles this by tracking dataflow. Whenever a new dataflow is encountered, Sentinel triggers profiling and makes migration decisions again.

Support of dynamic migration interval length. Sentinel uses the same length for all migration intervals. An alternative approach is to use different lengths for different intervals. Using such a dynamic interval length is helpful to avoid Cases 2 and 3. However, this method brings minimal performance benefit in practice, because Cases 2 and 3 do not happen often (see Table 4.3). Also, determining appropriate dynamic migration interval lengths has to explore a large search space of migration interval length, causing larger runtime overhead.

4.5 Applying Sentinel to GPU

Sentinel can be applied to HM on CPU; With slight extension, it can also be applied to address memory capacity limitation on GPUs by treating GPU’s global memory and CPU’s main memory as fast and slow memories respectively. We name Sentinel for GPU, *Sentinel-GPU*.

Profiling method. GPU typically uses a proprietary driver that we cannot modify to trigger protection faults to enable tensor-level profiling as for CPU. Although there is an open-source GPU driver [145], it supports limited types of GPU and is not stable; Although recent GPU has a paging mechanism to trigger traditional page faults [146], it cannot be used to count memory accesses on GPU, once pages are loaded into GPU memory; A binary instrumentation tool such as NVBit [211] or compiler tool can instrument load/store instructions but cannot directly measure main memory accesses. Hence, there is no tool to count memory accesses at page or tensor level for GPU. Also, introducing new hardware counters to collect page access statistics is possible, but hardware modification is expensive and unscalable.

To address the above problem, we use a customized pinned memory mechanism to enable tensor-level profiling for GPU. The traditional pinned memory mechanism allows GPU to access pages resident on CPU memory. By allocating tensors on pinned memory on CPU, we can use the existing profiling mechanism in Sentinel to count memory accesses from GPU. In particular, whenever GPU accesses a pinned memory page on CPU, a protection fault is triggered on CPU and handled by the fault handler in Sentinel to count it. Using the above method, we do not lose accuracy of counting memory accesses on GPU, because protection faults are caused by memory accesses on GPU, not on CPU.

However, implementing the above idea faces a challenge. The traditional pinned memory

mechanism disables paging, such that when GPU accesses a page resident on CPU memory, the page is guaranteed to be there. The implementation of this mechanism includes using the system call *mlock()* to lock PTE. As a result, Sentinel cannot modify the specific bit (bit 51) to trigger protection fault.

To address the above problem, we customize the pinned memory mechanism. In particular, Sentinel intercepts *mlock()* and bypasses it. To disable paging to ensure the correctness of the pinned memory mechanism, Sentinel temporarily disables OS-level page swapping mechanisms during the profiling using the existing system calls. This does not lock PTE, and hence allows Sentinel to set the bit to trigger protection faults.

Sentinel uses the above customized mechanism during the profiling, but must revert to the traditional GPU memory allocation and accesses in TensorFlow to avoid expensive CPU memory accesses. This reversion is feasible for those tensors that are repeatedly allocated and freed across training steps, but not possible for a few tensors that are allocated before all training steps and freed after them. For each of those tensors, Sentinel creates two copies, one using pinned memory and accessed during profiling, while the other using the traditional GPU memory allocation and used after profiling. Creating two copies does not require the user to change the implementation of DNN training, because it can be done by pointer switch through the runtime implementation. The two copies need to be synchronized after profiling to ensure the remaining training uses the most updated tensors. This synchronization overhead is paid in only one training step and ignorable in the whole training. We quantify this overhead in Section 4.7.3.

Handling Case 3. In Case 3, tensor migration cannot finish in time, because of lack of time for migration. A possible solution to handle this case on CPU-based HM is to leave tensors in slow memory on CPU. On GPU, however, the tensors must be placed on GPU memory when GPU accesses them (accessing CPU memory is too slow). Hence, there is no need to use the test-and-trial algorithm to handle Case 3. Handling this case must wait for tensor migration to complete, but subject to the optimization goal in Equation 4.2.

4.6 Implementation

We implement Sentinel in Linux v5.6.0 and TensorFlow v1.14. We change OS kernel for memory profiling; We change the TensorFlow runtime system for page migration (Figure 4.6). Sentinel introduces three APIs to trigger/stop memory profiling and identify DNN layers, which are *start_profile()*, *end_profile()*, and *add_layer()*. *start_profile()* triggers a system call to enable tracking of main memory accesses, memory (de)allocation to record lifetime of tensors.

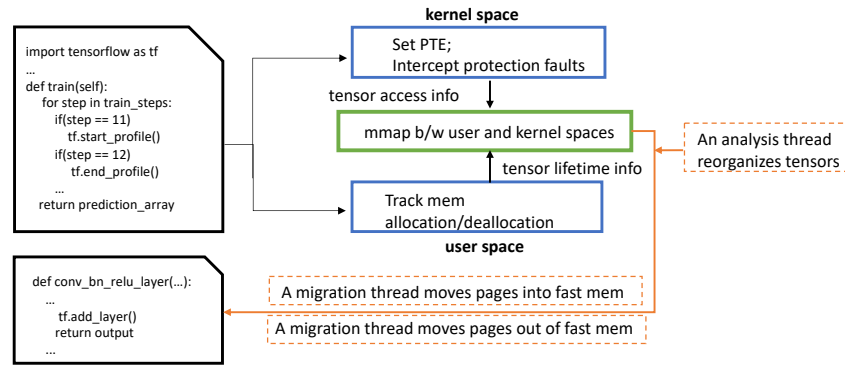


Figure 4.6: Implementation overview.

`add_layer()`, placed at the end of each layer, informs the runtime of where is each layer to determine migration intervals. Adding `start_profile()` and `end_profile()` includes only two lines of changes to the DNN model. Adding `add_layer()` includes 10-100 lines, depending on how many layers there are in a DNN model.

Figure 4.6 shows implementation details. Sentinel skips the first 10 training steps used by TensorFlow to detect hardware configurations, and uses the 11th for profiling. During the profiling, Sentinel collects memory access and lifetime information for each tensor from OS and TensorFlow respectively. After the profiling phase, Sentinel uses three helper threads: one for information analysis to determine migration intervals and make migration decision, one for data migration from fast to slow memory, and one for migration in the opposite way. The two migration threads work in parallel to accelerate migration. Sentinel uses the Linux system call `move_pages()` to migrate pages. Sentinel extends TensorFlow memory allocation and free functions by adding the customized data reorganization policy. Before the training happens, tensor are allocated in slow memory. After collecting the profiling results, Sentinel manages tensor allocation and migration.

GPU implementation. Similar to Sentinel, Sentinel-GPU leverages the APIs to enable online profiling and tensor management. To manipulate tensor allocation, Sentinel-GPU intercepts TensorFlow GPU memory allocators (such as `AllocateRaw` and `gpu_bfc_Allocator`), similar to [158]. Sentinel-GPU replaces those allocators with the customized ones for pinned memory control or tensor collocation. `add_layer()` is implemented as a CUDA kernel to execute at the end of each DNN layer. For short-lived tensors, Sentinel-GPU manages a memory pool allocated by `cudaMalloc` and enforces data reorganization. For long-lived tensors, Sentinel-GPU triggers bi-direction tensor movement by using CUDA events and streams. In particular, Sentinel uses two CUDA streams: one for computation and the other for tensor movement. Sentinel inserts tensor

Table 4.2: Hardware overview of experimental system.

Optane-based HM	
CPU	An Intel Xeon Gold 6252 CPU @2.30GHz
Last Level Cache	36608KB
Fast Memory	DDR4 DIMM: 96GB
Slow Memory	Optane DC PMM: 756GB
GPU-based HM	
GPU	Nvidia V100 with 16GB with 15.75 GB of GDDR6
CPU	Intel(R) Xeon(R) E5-2670 with 128 GB of DDR4
Interconnect	PCIe 3.0×16

movement events into the stream based on the decision on the migration intervals. Sentinel-GPU achieves asynchronous tensor movement with *cudaMemPrefetchAsync*.

4.7 Experimental Results

4.7.1 Experimental setup

We evaluate Sentinel on two HM platforms. One, named Optane-based HM, uses DRAM and Intel Optane DC persistent memory (PMM) as fast and slow memories respectively on CPU; The other, named GPU-based HM, treats GPU global memory and CPU main memory as fast and slow memories respectively. Table 4.2 gives details. PMM has two operating modes, *Memory Mode* and *App-direct Mode*. In Memory Mode, DRAM works as a hardware-managed cache to PMM. Running the application in this mode does not require modifications to the application. App-direct Mode allows programmer to explicitly control memory accesses to PMM and DRAM. Sentinel works in App-direct Mode and beats Memory Mode for large model training.

We evaluate five DNN models with small and large batch sizes (Table 4.3). We use the implementations of LSTM and MobileNet from TensorFlow [10], ResNet from [6], Bert from [9], and DCGAN from [5]. We use the default precision setting (FP32 or FP16) for floating point numbers. We report training throughput when the execution time per step becomes constant (usually after the first couple of steps).

4.7.2 Sentinel on Optane-based HM

Evaluation methodology. We compare Sentinel with a state-of-the-art memory management solution for HM on CPU [234]. This solution is based on a FIFO-based active list, and we name it *improved active list* (IAL). We also compare Sentinel with AutoTM [77]. AutoTM uses Integer Linear Programming (ILP) to decide tensor movement and placement based on static profiling and

nGraph compiler. To enable fair comparison, we implement the AutoTM’s memory management solution in TensorFlow. We compare Sentinel with IAL and AutoTM using the same fast memory size, which is 20% of the peak memory consumption of DNN models. This setting follows previous work [77, 234]. In addition, we compare Sentinel with the default NUMA allocation policy (first-touch NUMA) and Memory Mode. In our platform, DRAM and PMM belong to two NUMA nodes.

Table 4.3: DNN model for evaluation. “Sen.”, stands for Sentinel.

DNN Model	Dataset (Batchsize)	Peak mem. (GB)		# of steps used in Sen.	
		w/o Sen.	w/ Sen.	profiling	test & trial
ResNet-32	CIFAR-10 (128)	14.10	14.19	1	3
BERT-large	CoLA (32)	35.39	35.51	1	3
LSTM	PTB(20)	11.12	11.25	1	0
DCGAN	MNIST (128)	15.68	15.79	1	3
MobileNet	CIFAR-10 (128)	14.15	14.26	1	1
ResNet-200	CIFAR-10 (4K)	99.73	100.01	1	7
BERT-large	CoLA (128)	133.59	133.90	1	3
LSTM	PTB (4K)	29.97	30.09	1	0
DCGAN	celebA (10K)	115.10	115.23	1	7
MobileNet	CIFAR-100 (4K)	142.07	142.59	1	7

Overall performance. Figure 4.7 shows performance of IAL, AutoTM and Sentinel normalized by that of slow memory-only system. We use DNN models with small batch sizes for evaluation, because IAL code cannot work well for large batch sizes, either due to segfault or more than 10x performance slowdown. The figure shows that performance difference between Sentinel and fast memory-only system (shown as the red horizontal line in the figure) is very small (no difference in DCGAN, and 9% difference on average), while IAL has 46% performance difference on average. Sentinel outperforms IAL by 37% on average (up to 56%). Sentinel outperforms AutoTM by 17% on average (up to 31%) because of the following reasons. First, all tensor movements in AutoTM between fast and slow memories are exposed to the critical path, which incurs runtime overhead. Second, AutoTM uses static profiling to conclude that the output of an operation should be placed into slow memory with negligible performance impact. This conclusion is not true when the output is large. Sentinel uses dynamic profiling and attempts to put all tensors needed by the upcoming operations into fast memory, hence avoiding the performance problem.

Table 4.4 reports the total size of migrated tensors in one training step using IAL, AutoTM and Sentinel. Sentinel has 85% and 32% more migrations (on average) than IAL and AutoTM respectively. Frequent migrations allow Sentinel to make best use of fast memory for performance. Those migrations are overlapped with training to hide overhead.

Figure 4.8 shows performance with large batch sizes for first-touch NUMA, Memory Mode,

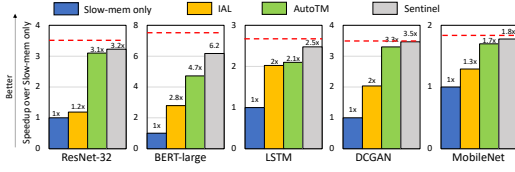


Figure 4.7: Performance speedup of IAL, AutoTM and Sentinel over slow-memory only. The red horizontal line shows performance of fast-memory only. Performance is normalized by that of slow-memory only.

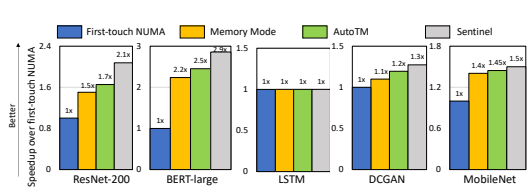


Figure 4.8: Performance with first-touch NUMA, Memory Mode, AutoTM and Sentinel, normalized by that of first-touch NUMA.

Table 4.4: Total size of migrated tensors in one training step.

	ResNet	BERT	DCGAN	LSTM	MobileNet
IAL	3.1GB	2.8GB	0.8GB	0.7GB	0.55GB
AutoTM	5.1GB	2.3GB	0.7GB	1.2GB	0.8GB
Sentinel	8GB	4GB	1.2GB	1.2GB	0.95GB

AutoTM, and Sentinel. Training with large batch sizes consumes large memory consumption (Table 4.3), creating challenges on data management in HM. The results are normalized by performance of first-touch NUMA. The results show that for the models whose peak memory consumption is larger than fast memory (e.g., ResNet200, BERT_large, DCGAN and MobileNet), Sentinel outperforms first-touch NUMA, Memory Mode and AutoTM by 1.7x, 1.2x and 1.1x (on average) respectively. For the models whose peak memory consumption is less than the fast memory size (LSTM), Sentinel has the same performance as first-touch NUMA, Memory Mode and AutoTM. In this case, DRAM is large enough to hold all tensors. This case shows ignorable overhead of Sentinel.

Memory bandwidth. We analyze memory bandwidth consumption in IAL and Sentinel, shown in Figure 4.9. Compared with IAL, Sentinel consumes much higher (7.3x on average) memory bandwidth in fast memory, indicating that fast memory accesses happen much more often in Sentinel than in IAL. Sentinel also has lower memory bandwidth consumption in slow memory, compared with IAL, indicating that Sentinel reduces accesses in slow memory.

Runtime overhead. Table 4.3 shows total number of training steps used for profiling and test-and-trial. Those steps have longer execution time than regular training steps, hence introducing runtime overhead. On average, Sentinel uses only 1.8 steps. Each of those steps is extended by up to 5x in terms of execution time. However, such overhead is amortized by millions of steps. As a result, the runtime overhead of Sentinel is negligible (less than 1%).

Memory overhead. Using Sentinel for tensor-level profiling increases peak memory consumption,

causing memory overhead. Table 4.3 shows peak memory consumption. Sentinel does not increase peak memory consumption much (by 2.4% at most). This is because tensors larger than one page dominate total memory consumption. Profiling those tensors with Sentinel does not cause memory overhead.

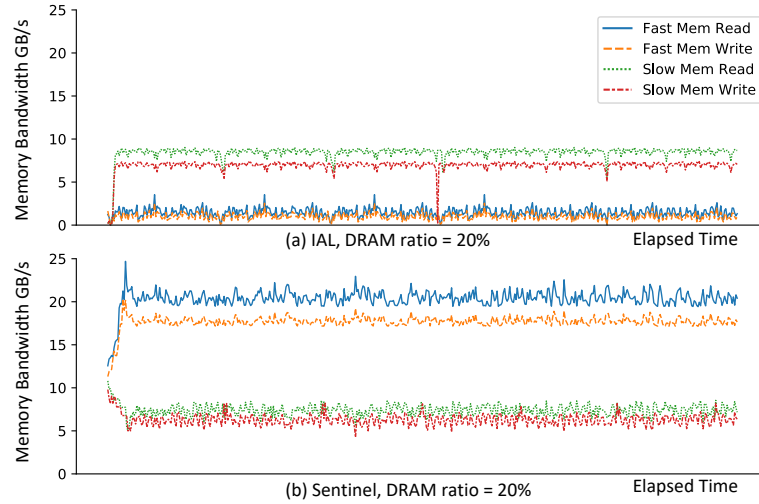


Figure 4.9: Memory access bandwidth during training of ResNet-32.

Sensitivity study. We change fast memory size and measure performance with small batches. Figure 4.10 shows the results. In general, larger fast memory gives better performance. When the fast memory size is 60% of peak memory consumption, all of DNN models on HM with Sentinel do not have any performance difference from the fast memory-only system. Also, with Sentinel, performance is not sensitive to fast memory size: There is at most 17% performance variance when fast memory size is changed from 20% to 40% of peak memory consumption. This demonstrates how Sentinel effectively uses tensor movement to make the best use of fast memory.

Saving fast memory size. Figure 4.7 shows using 20% of peak memory consumption of DNN models as fast memory size, Sentinel on HM has similar performance (9% difference on average) as the fast memory-only. This brings 80% saving in fast memory. Figure 4.10 shows using 60% of peak memory consumption as fast memory size, no performance loss.

To further study Sentinel’s effectiveness, we use ResNet with various topology and peak memory consumption. We report the minimum fast memory size with which Sentinel performs the same as the fast memory-only. Figure 4.11 shows peak memory consumption and fast memory size for all ResNet variants. The figure shows that although peak memory consumption increases quickly as ResNet becomes more complicated, the fast memory size increases in a much slower

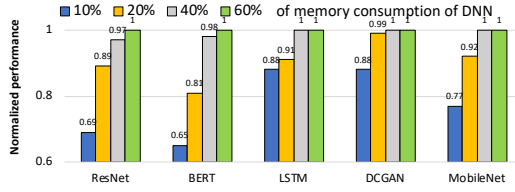


Figure 4.10: Performance with Sentinel under various sizes of fast memory. The fast memory size is shown as the percentage of peak memory consumption of DNN models. Performance is normalized by that of the fast memory-only.

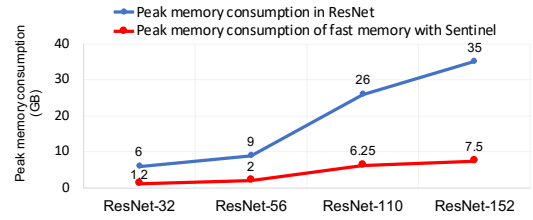


Figure 4.11: Comparison between peak memory consumption of DNN models and fast memory size for ResNet variants.

rate because of adaptive layer-based migration. This demonstrates the effectiveness of using Sentinel to save fast memory size.

4.7.3 Sentinel on GPU-based HM

Evaluation methodology. We use Nvidia Tesla V100 GPU shown in Table 4.2 with CUDA v10.1 for evaluation. We compare Sentinel-GPU with five existing work on GPU, including Unified Memory (UM) [146], vDNN [182], AutoTM [77], SwapAdvisor [79] and Capuchin [158]. UM automatically moves tensors from CPU to GPU in the event of a GPU page fault, and moves least-used pages from GPU to CPU. vDNN is a solution using GPU-based HM for DNN training. vDNN focuses on convolution layers and migrates input tensors of convolution layers between CPU and GPU memories; vDNN tries to overlap the migration of the input tensors with convolution computation. AutoTM works for both CPU and GPU, and uses ILP to decide tensor movement and placement. We implemented asynchronous tensor migration in AutoTM as described in [77]. SwapAdvisor uses the Generic Algorithm (GA) to find a good combination of memory allocation and operation scheduling on MXNet [37]. Capuchin is a state-of-the-art solution using GPU-based HM for DNN training. Capuchin overlaps tensor movement with training. When the tensor movement overhead is too large to be overlapped, Capuchin discards tensors and recomputes them when needed to save GPU memory. vDNN, SwapAdvisor, AutoTM and Capuchin are either close-sourced or not implemented on TensorFlow. We implement their migration strategies in TensorFlow.

Profiling method. We make the comparison in terms of profiling method. UM does not use any profiling, and uses on-demand tensor movement. As a result, UM causes large runtime

overhead because most of tensor movement is exposed to the critical path. vDNN does not use any profiling, but heavily rely on domain knowledge to decide tensor migration. As a result, vDNN only works for specific models (feedforward CNN models) and cannot handle recursive structures in DNN models. AutoTM collects execution time of individual operations at compilation time. Such static profiling method misleads tensor migration, when the batch size or hardware used in production is different from the ones used in static profiling. SwapAdvisor uses a lot of training steps for dynamic profiling. Because of the GA algorithm SwapAdvisor uses to decide tensor migration, SwapAdvisor suggests to use 30 minutes to make the final migration decision at runtime [79]; According to our experimental results, for a large model such as BERT-large, SwapAdvisor cannot make the decision within 30 minutes. This decision process is too slow for some model training (e.g., NLP fine-tuning which can take less than two hours [9]). Capuchin and Sentinel use dynamic profiling, whose overhead is negligible (a few seconds and less than 1%).

Maximum batch size. We compare vDNN, AutoTM, SwapAdvisor, Capuchin and Sentinel-GPU in terms of maximum batch size each solution can achieve, given the same GPU memory capacity. This comparison aims to show how effectively these solutions save GPU memory. We do not evaluate UM, because its maximum batch size is limited by CPU memory and can be much larger than that with vDNN and Sentinel-GPU, but with much worse performance (shown in Figure 4.12). Table 4.5 shows the result. The result for “TensorFlow” in Table 4.5 is collected without using tensor migration. Compared with TensorFlow without tensor migration, Sentinel-GPU increases batch size by 4.18x on average. vDNN is designed for feedforward CNN models and cannot handle recursive structures in a DNN graph. Hence it cannot work for LSTM and BERT-large. For CNN models, Sentinel-GPU outperforms vDNN by 1.9x. This is because Sentinel-GPU migrates tensors as many as possible to save GPU memory, whereas vDNN only focuses on input tensors of the convolution layers. Sentinel-GPU outperforms SwapAdvisor by 1.1x. This is because SwapAdvisor aims to minimize training time instead of minimizing memory consumption of tensors in GPU. AutoTM, Capuchin and Sentinel-GPU achieve a comparable maximum batch size, because all of them try to migrate tensors out of GPU memory as much as possible. They differ in training throughput, discussed as follows.

Training throughput. For each model, we use three batch sizes, shown in Figure 4.12. Throughput in Figure 4.12 is normalized by that of UM. With the largest batch size shown in Figure 4.12, Figure 4.13 shows the performance for two components (migration overhead in the critical path and recomputation) in one training step for deeper analysis, and percentage numbers

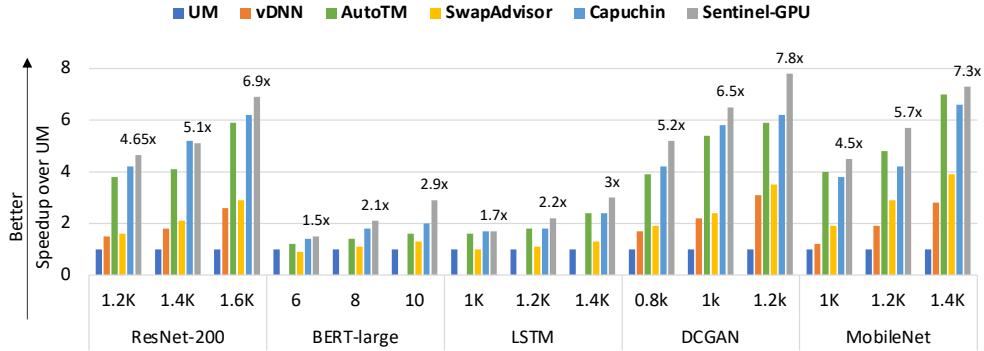


Figure 4.12: Performance of UM, vDNN, AutoTM, SwapAdvisor, and Capuchin and Sentinel-GPU, normalized by that of UM. vDNN cannot work for BERT and LSTM.

on top of bars in Figure 4.13 are the ratio in terms of execution time of one training step. For Sentinel-GPU, Figure 4.13 shows performance breakdown results to quantify the contributions of various techniques in Sentinel-GPU. “Direct tensor migration” does not use migration interval and trigger tensor migration simply based on tensor forthcoming usage; It does not reserve space for short-lived tensors; “w/ det. MI” uses an optimal migration interval length but without space reservation; “w/all” is the full featured Sentinel.

UM vs. Sentinel-GPU. Sentinel-GPU has 1.1x-7.8x higher throughput than UM. Such a large performance gain comes from effectively prefetching tensors from CPU to GPU and reduction of migration overhead in Sentinel-GPU.

vDNN vs. Sentinel-GPU. For CNN models, Sentinel-GPU outperforms vDNN by 2x. Similar to Sentinel-GPU, vDNN tries to overlap tensor movement with computation. However, vDNN does not consider time difference between layers, which exposes most of tensor migration overhead to critical path (3x more than with Sentinel-GPU).

SwapAdvisor vs. Sentinel-GPU. Sentinel-GPU outperforms SwapAdvisor by 65% on average (up to 110%). The GA in SwapAdvisor is too slow (more than 30 minutes) to find an optimal solution for tensor placement and migration. The process of finding the solution causes slowdown; The tensor migration overhead in SwapAdvisor is 81% larger than that in Sentinel-GPU, because SwapAdvisor fails to hide that.

AutoTM vs. Sentinel-GPU. Sentinel-GPU outperforms AutoTM by 17% on average (up to 29%). Sentinel-GPU with the space reservation reduces migration overhead by 8% of training time, compared with AutoTM, because this technique avoids unnecessary tensor movement while AutoTM exposes tensor movement into the critical path. Avoiding page-level false sharing in Sentinel-GPU contributes additional 9% benefit over AutoTM.

Table 4.5: Maximum Batch Size with vDNN, AutoTM, SwapAdvisor, Capuchin and Sentinel-GPU

	ResNet-200 (CIFAR10)	BERT-large (CoLA)	LSTM (PTB)	DCGAN (celebA)	MobileNet (CIFAR100)
TensorFlow	1K	5	800	0.6K	0.8K
vDNN	4.2K	not work	not work	1.4K	1.2K
AutoTM	5.6K	27	1.4K	2.5K	3.2K
SwapAdvisor	5.4K	25	1.2K	2.4K	3.1K
Capuchin	5.9K	27	1.4K	2.7K	3.2K
Sentinel-GPU	5.7K	28	1.5K	2.5K	3.2K

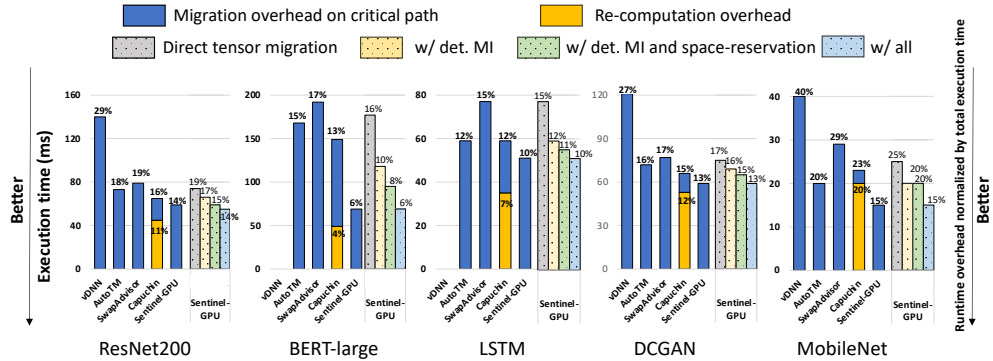


Figure 4.13: Performance breakdown for vDNN, AutoTM, SwapAdvisor, Capuchin and Sentinel-GPU. “det. MI” stands for “determine an appropriate migration interval length”. Percentage numbers on top of bars are the ratio in terms of execution time of one training step.

Capuchin vs. Sentinel-GPU. Sentinel-GPU outperforms Capuchin by 16% on average (up to 21%). Because of the pervasiveness of page-level false sharing, Sentinel-GPU improves performance by 11% - 21%. In Capuchin, recomputation takes about 11% of the training time while Sentinel-GPU does not have recomputation overhead. As a result, although the migration time in Capuchin is shorter than in Sentinel-GPU, the net effect is that Sentinel-GPU outperforms Capuchin.

4.8 Related Work

Comparison with recent efforts on using HM for DNN training. We review and evaluate them in Sections 4.2 and 4.7.

Using Managed Runtime for Data Management on HM. Existing efforts [16, 17, 216, 228] leverage managed runtime such as JVM. There are two differences between them and Sentinel. (1) The existing efforts couple data migration with garbage collection, and hence miss opportunities to minimize data migration overhead; (2) The existing efforts do not proactively migrate data

objects to save fast memory space.

Page-based Runtime Data Management on HM. Existing proposals [14, 78, 95, 121, 181, 226, 227, 234] explore various page placement policies based on memory access profiling. Some works [14, 78, 95] track page accesses by setting and resetting PTE as Sentinel does, but this tracking mechanism incurs high runtime overhead. Unlike the above work, Sentinel leverages DNN domain knowledge, and hence only profiles a small portion of total execution (one training step) without paying large runtime overhead and losing accuracy. Also, Sentinel associates page-level profiling results with tensors, making profiling results more meaningful for tensor migration.

4.9 Conclusions

Training DNN faces a problem on memory capacity. This paper focuses on how to use HM to address this problem without losing training throughput while saving fast memory. We introduce a runtime system (Sentinel) based on a unique and comprehensive performance study on all tensors in various linear and nonlinear models. The runtime system is featured with a novel tensor-level profiling method and runtime techniques to improve tensor migration efficiency for high performance and saving fast memory capacity. Evaluating on Optane-based HM and CPU-GPU-based HM, we show Sentinel outperforms seven software- and hardware-based solutions.

Chapter 5

Democratizing Billion-Scale Model Training

Large-scale model training has been a playing ground for a limited few users, because it often requires complex model refactoring and access to prohibitively expensive GPU clusters. ZeRO-Offload changes the large model training landscape by making large model training accessible to nearly everyone. It can train models with over 13 billion parameters on a single GPU, a 10x increase in size compared to popular framework such as PyTorch, and it does so without requiring any model change from data scientists or sacrificing computational efficiency.

ZeRO-Offload enables large model training by offloading data and compute to CPU. To preserve compute efficiency, it is designed to minimize data movement to/from GPU, and reduce CPU compute time while maximizing memory savings on GPU. As a result, ZeRO-Offload can achieve 40 TFlops/GPU on a single NVIDIA V100 GPU for 10B parameter model, compared to 30TF using PyTorch alone for a 1.4B parameter model, the largest that can be trained without running out of memory on GPU. ZeRO-Offload is also designed to scale on multiple-GPUs when available, offering near-linear speedup on up to 128 GPUs. Additionally, it can work together with model parallelism to train models with over 70 billion parameters on a single DGX-2 box, a 4.5x increase in model size compared to using model parallelism alone.

By combining compute and memory efficiency with ease-of-use, ZeRO-Offload democratizes large-scale model training making it accessible to even data scientists with access to just a single GPU.

5.1 Introduction

Since the advent of the attention-based deep learning (DL) models in 2017, we have seen an exponential growth in DL model size, fueled by substantial quality gains that these attention based models can offer with the increase in the number of parameters. For example, the largest language model in literature had less than 100M parameters in 2017. It grew to over 300M with BERT [43] in 2018, and increased to tens of billions in 2019 with models such as GPT-2 [34], T5 [171], Megatron-LM [195] and Turing-NLG [184]. Today, the largest language model GPT-3 [33] has a staggering number of 175B parameters. With the three orders of magnitude growth in model size since 2017, the model accuracy continues to improve with the model size [97]. Recent studies in fact show that larger models are more resource-efficient to train than smaller ones [97] for a given accuracy target. As a result, we expect the model size to continue growing in the future.

However, accessibility to large model training is severely limited by the nature of state-of-art system technologies. Those technologies make entry into the large model training space prohibitively expensive. To be more specific, distributed parallel DL training technologies such as pipeline parallelism [81], model parallelism [195], and ZeRO [172] (Zero Redundancy Optimizer) allow transcending the memory boundaries of single GPU/accelerator device by splitting the model states (parameters, gradients and optimizer states) across multiple GPU devices, enabling massive models that would otherwise simply not fit in a single GPU memory. All record-breaking large models such as GPT-2, Megatron-LM, Turing-NLG, and GPT-3, were trained using a combination of the aforementioned technologies. However, all of these DL parallel technologies require having enough GPU devices such that the aggregated GPU memory can hold the model states required for the training. For example, training a 10B parameter model efficiently requires a DGX-2 equivalent node with 16 NVIDIA V100 cards, which costs over 100K, beyond the reach of many data scientists, and even many academic and industrial institutions.

Heterogeneous DL training is a promising approach to reduce GPU memory requirement by exploiting CPU memory. Many efforts have been made in this direction [77, 79, 92, 158, 178, 182, 212, 218, 244]. Nearly all of them target CNN based models, where activation memory is the memory bottleneck, and model size is fairly small (less than 500M). However, the primary memory bottleneck for recent attention based large model training are the model states, instead of activation memory. There is an absence in literature studying these workloads for heterogeneous DL training. Additionally, existing efforts on heterogeneous training are further limited in two major ways: i) nearly all of them exploit CPU memory, but not CPU compute, which we show can

be used to significantly reduce the CPU-GPU communication overhead, and ii) they are mostly designed for and evaluated on single GPU [79, 92, 178, 244], without a clear path to scaling efficiently on multiple GPUs, which is crucial for large model training.

Addressing the aforementioned limitation, we attempt to democratize large model training by developing ZeRO-Offload, a novel heterogeneous DL training technology designed specifically for large model training. ZeRO-Offload exploits both CPU memory and compute for offloading, while offering a clear path towards efficiently scaling on multiple GPUs by working with ZeRO-powered data parallelism [172]. Additionally, our first principle analysis shows that ZeRO-Offload provides an optimal and the only optimal solution in maximizing memory saving while minimizing communication overhead and CPU compute overhead for large model training.

ZeRO-Offload is designed around three main pillars: i) Efficiency, ii) Scalability, and iii) Usability.

Efficiency: The offload strategy in ZeRO-Offload is designed with the goal of achieving comparable compute efficiency to the state-of-art non-offload strategies but for significantly larger models. To achieve this goal, we rely on first principle analysis to identify a *unique optimal* computation and data partitioning strategy between CPU and GPU devices. This strategy is optimal in three key aspects: i) it requires orders-of-magnitude fewer computation on CPU compared to GPU, preventing the CPU compute from becoming a performance bottleneck, ii) it minimizes the communication volume between CPU and GPU preventing communication from becoming a bottleneck, and iii) it provably maximizes memory savings on GPU while achieving minimum communication volume.

Our analysis shows that to be optimal in the aforementioned regards, we must offload the gradients, optimizer states and optimizer computation to CPU, while keeping the parameters and forward and backward computation on GPU. This strategy enables a 10x increase in model size, with minimum communication and limited CPU computation, which allows us to train 13B parameters on a single NVIDIA V100 GPU at 40 TFLOPS, compared to 30 TFLOPS on the same GPU with 1.2B parameters, the largest model that can be trained without any CPU offloading.

Offloading optimizer computation requires CPU to perform $O(M)$ computation compared to $O(MB)$ on GPU where M and B are the model size and batch sizes respectively. In most cases, the batch size is large, and CPU computation is not a bottleneck, but for small batch sizes, the CPU compute can be a bottleneck. We address this using two optimizations: i) an efficient *CPU optimizer* that is up to 6x faster than the-state-of-art, and ii) One-step *delayed parameter update* that allows overlapping the CPU optimizer step with GPU compute, while preserving accuracy.

Together, they preserve efficiency for ZeRO-Offload even with small batch sizes.

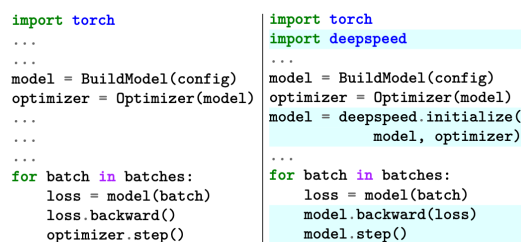
Scalability: Good scalability is crucial to take advantage of multiple GPUs that may be available to some data scientists. In the DL community, data parallelism is generally used as the de facto standard to scale DL training to multiple GPUs [42, 190, 249]. However, it is not designed to work with heterogeneous training and presents scalability challenges because of the replication of data and computation in data parallel training. Data parallel training replicates all the model states such as optimizer states, parameters, and gradients, and it also replicates the optimizer computation on each GPU. Therefore, offloading model states or optimizer computation to CPU in combination with data parallelism will result in significant replication of communication and CPU compute: increase the CPU memory requirement proportionally to the data parallelism degree while limiting throughput scalability due to the increased communication.

To address these challenges, ZeRO-Offload combines unique optimal offload strategy with ZeRO [172] powered data parallelism instead of traditional data parallelism. The symbiosis allows ZeRO-Offload to maintain a single copy of the optimizer states on the CPU memory regardless of the data parallel degree. Furthermore, it keeps the aggregate communication volume between GPU and CPU, as well as the aggregate CPU computation a constant regardless of data parallelism, allowing ZeRO-Offload to effectively utilize the linear increase in CPU compute with the increase in the data parallelism degree. As a result, ZeRO-Offload achieves excellent scalability on up to 128 GPUs.

In addition to working with ZeRO powered data parallelism, ZeRO-Offload can be combined with model parallelism [192, 195] to achieve higher memory savings, when multiple GPUs are available.

Usability: ZeRO-Offload is available as part of an OpenSource PyTorch library, DeepSpeed (www.deepspeed.ai). Unlike most strategies discussed in Section 5.2, ZeRO-Offload does not require model refactoring to work. In fact, PyTorch users can enable ZeRO-Offload with few lines of code change to their existing training pipeline as shown in Figure 5.1, allowing to train 10x larger models easily.

Contributions. To the best of our



```

import torch
...
...
model = BuildModel(config)
optimizer = Optimizer(model)
...
...
for batch in batches:
    loss = model(batch)
    loss.backward()
    optimizer.step()

```

```

import torch
import deepspeed
...
...
model = BuildModel(config)
optimizer = Optimizer(model)
model = deepspeed.initialize(
    model, optimizer)
...
...
for batch in batches:
    loss = model(batch)
    model.backward(loss)
    model.step()

```

Figure 5.1: ZeRO-Offload can be enabled with a few lines of change. The code on left shows a standard training pipeline, while the right shows the same pipeline with ZeRO-Offload enabled.

knowledge, ZeRO-Offload is the first fully distributed all-reduced based training framework using CPU memory and computation resources to train large-scale models. We summarize contributions are as follows:

- A unique optimal offload strategy for heterogeneous large model training on GPU + CPU system that enables 10x larger model on a single GPU without sacrificing efficiency (Sec. 5.3 and Sec. 5.4.1).
- Highly scalable multi-GPU design through i) a symbiotic combination of offload strategy with ZeRO powered data parallelism (Sec. 5.4.2), allowing ZeRO-Offload to achieve near-linear scalability, and ii) seamless integration with model-parallel training [195], enabling even larger models than using ZeRO-Offload or model parallelism alone (Sec. 5.4.2).
- Open-source implementation of ZeRO-Offload in PyTorch.
- Extensive evaluation demonstrating i) *Model Scale*: 10x increase in model size with up to 13B on a single GPU and 4x increase in model size over model parallelism with up to 70B parameters on a DGX-2 node. ii) *Efficiency*: Over 40 TFlops for a 10B parameters on a single NVIDIA V100, compared to 30 TFLOPS on the same GPU with 1.4B parameters, the largest model that can be trained without any CPU offloading; Outperform two state-of-the-art heterogeneous DL training frameworks by 22% and 37% respectively on a single GPU. iii) *Scalability*: Near-perfect linear scalability for a 10B parameter model on up to 128 GPUs. iv) CPU overhead reduction with our ADAM implementation with 6x speedup over PyTorch optimizer and up to 1.5X improvement in end-to-end throughput with delayed parameter update optimizations (Sec. 5.6).

5.2 Background and Related Work

Memory consumption in large model training. The full spectrum of memory consumption during DL model training can be classified into two parts: i) model states and ii) residual states [172]. Model states include parameters, gradients, and optimizer states (such as momentum and variances in Adam [99]); Residual states include activations, temporary buffers, and unusable fragmented memory.

Model states are the primary source of memory bottleneck in large model training. We consider the memory consumption due to model states for large transformer models such as Megatron-LM (8 billion) [195], T5 (11 billion) [171], and Turing-NLG [184] (17.2 billion). They are trained with float-16 mixed precision training [144] and Adam optimizer [99].

Mixed precision training often keeps two copies of the parameters, one in float-16 (fp16) and

the other in float-32 (fp32). The gradients are stored in fp16. In addition to the parameters and gradients, the Adam optimizer keeps track of the momentum and variance of the gradients. These optimizer states are stored in fp32. Therefore, training a model in mixed precision with the Adam optimizer requires at least 2 bytes of memory for each fp16 parameter and gradient, and 4 byte of memory for each fp32 parameter, and the momentum and variance of each gradient. In total, a model with M parameters requires $16 \times M$ bytes of memory. Therefore, the model states for Megatron-LM, T5 and Turing-NLG require 128 GB, 176 GB and 284 GB, respectively, which are clearly beyond the memory capacity of even the current flagship NVIDIA A100 GPU with 80 GB of memory.

Significant amount of work has been done in the recent years to enable large model training, which requires more memory than what is available on a single GPU to fit these model and residual states. These efforts can be classified broadly into two categories: i) scale-out training and ii) scale-up training based approaches. We discuss them as follows.

Scale out large model training. Scale-out training uses aggregate memory of multiple GPUs to satisfy the memory requirement for large model training. Two prominent examples of scale out training is model parallelism [42, 195] and pipeline parallelism [72, 81], both partitioning the model states and the residual states across multiple GPUs. Model parallelism [42, 195] partitions the model vertically and distributes the model partitions to multiple GPU devices in order to train large models. Pipeline parallelism [72, 81] on the other hand parallelizes model training by partitioning the model horizontally across layers. Both of these approaches must change the user model to work, therefore can limit usability.

A recent work, ZeRO [172], provides an alternative to model and pipeline parallelisms to train large models. ZeRO splits the training batch across multiple GPUs similar to data parallel training [42, 190, 249], but unlike data parallel training which replicates all the model states on each GPU, ZeRO partitions them across all GPUs, and uses communication collectives to gather individual parameters as needed during the training. ZeRO does not require changes to the user model to work, making it more generic than model or pipeline parallel training. It also offers better compute efficiency and scalability.

Despite the ability of model and pipeline parallelisms, and ZeRO to train large models, they all require multiple GPUs such that the aggregate GPU memory can hold the model and residual states for training large models. In contrast, ZeRO-Offload is designed to fit a larger model by offloading model states to CPU memory and can train a 10x larger model on a single GPU without sacrificing efficiency. When multiple GPUs are available, ZeRO-Offload is designed to work

together with ZeRO to offer excellent scalability, or in conjunction with model parallelism to fit even larger model sizes that is not possible with ZeRO-Offload or model parallelism alone.

Scale up large model training. Existing work scales up model size in a single GPU through three major approaches. The first approach trades computation for memory saving from activations (residual memory) by recomputing from checkpoints [38]. The second approach uses compression techniques such as using low or mixed precision [144] for model training, saving on both model states and activations. The third approach uses an external memory such as the CPU memory as an extension of GPU memory to increase memory capacity during training [77, 79, 92, 158, 178, 182, 218].

Our work, ZeRO-Offload falls under the third approach. Unlike ZeRO-Offload, the above efforts only offload data to CPU but not compute, and they use smaller models training. Furthermore, none of the above works is communication optimal, leading to extra communication between CPU and GPU and hurting training throughput. In contrast, a recent work called L2L [164] can enable multi-billion parameter training by managing memory usage in GPU layer by layer. In particular, L2L synchronously moves tensors needed in the upcoming layer into GPU memory for computation and keeps the rest of tensors into CPU memory for memory saving. In comparison to ZeRO-Offload, it offers limited efficiency due to extra communication overhead, does not offer a way to scale out across devices, and requires model refactoring, making it difficult to use.

ZeRO powered data parallel training. ZeRO-Offload works with ZeRO to scale DL training to multiple GPUs. ZeRO has three stages, ZeRO-1, ZeRO-2 and ZeRO-3 corresponding to the partitioning of the three different model states, optimizer states, gradients and parameters, respectively. ZeRO-1 partitions the optimizer states only, while ZeRO-2 partitions gradients in addition to optimizer states, and ZeRO-3 partitions all model states. ZeRO-Offload works symbiotically with ZeRO-2, and therefore we discuss it further.

In ZeRO-2, each GPU stores a replica of all the parameters, but only updates a mutually exclusive portion of it during the parameter update at the end of each training step. As each GPU only updates a portion of the parameters, they only store optimizer states and gradients required to make that update. After the update, each GPU sends its portion of the updated parameters to all the other GPUs using an `all-gather` communication collective. ZeRO-2 computation and communication schedule is described below:

During the forward pass, each GPU computes the loss with respect to a different mini-batch. During the backward propagation, as each gradient is computed, it is averaged using a `reduce` operator at the GPU/GPUs that owns the gradient or part of the gradient. After the backward pass,

each GPU updates its portion of the parameters and optimizer states using the averaged gradients corresponding to that portion. After this, an `all-gather` is performed to receive the rest of the parameter update computed on other GPUs.

5.3 Unique Optimal Offload Strategy

ZeRO-Offload is designed to enable efficient large model training on a single or multiple GPUs by offloading some of the model states from GPU to CPU memory during training. As discussed in Sec. 5.2, model states: parameters, gradients, and the optimizer states, are the primary source of memory bottleneck in large model training. By offloading some of these model states to CPU, ZeRO-Offload can enable training of significantly larger models¹. However, identifying the optimal offloading strategy is non-trivial. There are numerous ways to offload model states to CPU memory, each with a different trade-off in terms of CPU computation, and GPU-CPU communication, both of which can limit the training efficiency.

To identify the optimal offload strategy, ZeRO-Offload models the DL training as data-flow graph and uses first principle analysis to efficiently partition this graph between CPU and GPU devices. ZeRO-Offload partitions the graph in a way that is *optimal* in three key aspects: i) it requires orders-of-magnitude fewer computation on CPU compared to GPU, which prevents CPU from becoming a performance bottleneck (Sec. 5.3.1), ii) it guarantees the minimization of communication volume between CPU and GPU memory (Sec. 5.3.3), and iii) it provably maximizes the memory savings while achieving minimum communication volume (Sec. 5.3.4). In fact, ZeRO-Offload can achieve high efficiency during training that is comparable to non-offload training and it is *unique optimal*, meaning no other solution can offer better memory savings without increasing the communication volume or increasing CPU computation.

In this section, we discuss the derivation of our unique optimal offload strategy. Our strategy is specifically designed for *mixed precision training with Adam optimizer* which is the de facto training recipe for large model training.

5.3.1 DL Training as a Data-Flow Graph

The DL training workload can be represented as a weighted directed graph of data and computation, as shown in Figure 5.2, where the circular nodes represents model states (parameter16,

¹ZeRO-Offload only offloads model states. Offloading secondary sources of memory bottleneck such as activation memory is beyond scope of our offload strategy. Given that they are significantly smaller than model states, we ignore them for the purpose of our analysis. Furthermore, the first and second approaches described in Sec. 5.2 can be used in conjunction with ZeRO-Offload to reduce activation memory

gradient16, parameter32, momentum32, variance32), and the rectangular nodes represents computation (forward, backward, param update). The edges in the graph represents the data flow between the nodes, and the weight of an edge is the total data volume in bytes that flows through it during any given training iteration. For a model with M parameters, the weight of the edges in this graph is either $2M$ where the source node produces fp16 model states, or $4M$ where the source node produces fp32 model states.

An offload strategy between GPU and CPU can be represented using a two-way partitioning of this graph, such that computation nodes in a partition would be executed on the device that owns the partition, and the data nodes in the partition will be stored on device that owns the partition. The total data volume that must be communicated between GPU and CPU is given by the weight of edges running across two partitions.

There are numerous ways to partition this graph. In the following sections, we use first principles to simplify the data flow graph to reduce the number of possible choices based on three different efficiency metric: i) CPU computation overhead, ii) communication overhead, and iii) memory savings.

5.3.2 Limiting CPU Computation

The CPU computation throughput is multiple orders of magnitude slower than the GPU computation throughput. Therefore, offloading large computation graph to CPU will severely limit training efficiency. As such, we must avoid offloading compute intensive components to the CPU.

The compute complexity of DL training per iteration is generally given by $O(MB)$, where M is the model size and B is the effective batch size. To avoid CPU computation from becoming a bottleneck, only those computations that have a compute complexity lower than $O(MB)$ should be offloaded to CPU. This means that the forward propagation and backward propagation both of which have a compute complexity of $O(MB)$ must be done on GPU, while remaining computations such as norm calculations, weight updates etc that have a complexity of $O(M)$ may be offloaded to CPU.

Based on this simple observation we fuse the forward and backward nodes in our data flow graph into a single super-node (FWD-BWD) and assign it to GPU.

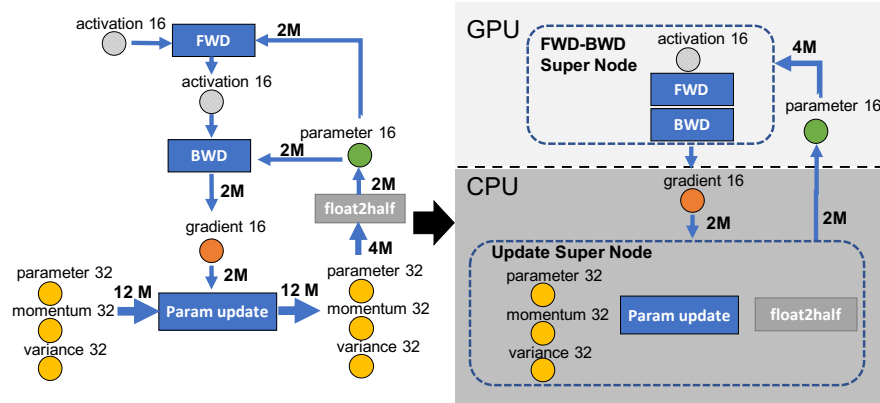


Figure 5.2: The dataflow of fully connected neural networks with M parameters. We use activation checkpoint to reduce activation memory to avoid activation migration between CPU and GPU.

5.3.3 Minimizing Communication Volume

The CPU memory bandwidth is at least an order of magnitude faster than the PCI-E bandwidth between CPU and GPU, while the GPU memory is another order of magnitude faster than even the CPU memory. Therefore, we must minimize the communication volume between CPU and GPU memory to prevent the PCI-E bandwidth from becoming a training performance bottleneck. To do so we must first identify the theoretical minimum communication volume for a model-state offload strategy.

The minimum communication volume for any model-state offload strategy is given by $4M$ ². Note that after fusing the forward and backward into a single super-node as discussed in Sec. 5.3.2, each node in our data flow graph is part of a cycle. Therefore, any partitioning of this graph would require cutting at least two edges, each of which has a edge weight of at least $2M$, resulting in a total communication of at least $4M$.

If we choose to limit the communication volume to this bare minimum, we can greatly simplify our data-flow graph and reduce the number of partitioning strategies to a handful:

Creating fp32 super-node. Notice that any partitioning strategy that does not co-locate the fp32 model states with their producer and consumer nodes cannot achieve the minimum communication volume of $4M$. Such a partition must cut at least one edge with a weight of $4M$, and the other with at least $2M$, resulting in a communication volume of at least $6M$. Therefore, to achieve the minimum communication volume, all offload strategies must co-locate fp32 model

²Please note that it is possible to reduce the communication volume further by only offloading partial model states. For simplification, we assume that an offload of a model state implies that we offload the entire model state. Our analysis on the memory savings per communication volume, still holds even if we offload partial model states

states with their producer and consumer operators, i.e., the fp32 model states (momentum32, variance32 and parameter32) must be co-located with the *Param Update* and the *float2half* computation.

This constraint allows us to treat all the aforementioned fp32 data and compute nodes in the data flow graph as a single super-node that we refer to as *Update Super*. We show this reduced data flow graph in Figure 5.2, consisting of only four nodes: *FWD-BWD Super* node, *p16* data node, *g16* data node, and *Update Super* node.

p16 assignment. To achieve the minimum communication volume, *p16* must be co-located with *FWD-BWD Super* because the edge weight between these two nodes is 4M. Separating these two nodes, would increase the communication volume to 6M (i.e., 4M + 2M). Since, we have already assigned node *FWD-BWD Super* to GPU to limit computation on CPU, *p16* must also be assigned to GPU.

5.3.4 Maximizing Memory Savings

After simplifying the data flow graph to minimize communication volume, only *g16* and *Update Super* remain to be assigned. Notice that at this point, all partitions will result in minimum communication volume, so we can prune the choices further to maximize the memory savings on GPU. Table 5.1 shows the memory savings of all valid partitioning strategies that minimize the communication volume. The maximum memory savings of 8x can be achieved by offloading both *g16* and *Update Super* to CPU.

Table 5.1: Memory savings for offload strategies that minimize communication volume compared to the baseline.

FWD-BWD	p16	g16	Update	Memory	Reduction
gpu	gpu	gpu	gpu	16M	1x (baseline)
gpu	gpu	cpu	gpu	14M	1.14x
gpu	gpu	gpu	cpu	4M	4x
gpu	gpu	cpu	cpu	4M	8x

5.3.5 A Unique and Optimal Offload Strategy

ZeRO-Offload allocates all the fp32 model states along with the fp16 gradients on the CPU memory, and it also computes the parameter updates on CPU. The fp16 parameters are kept on GPU and the forward and backward computations are also done on GPU.

We arrive at this offload strategy by simplifying our data flow graph and eliminating all other partitioning strategies as they do not limit CPU computation, minimize communication volume, or maximize memory savings. Therefore, ZeRO-Offload is not only optimal in terms of the

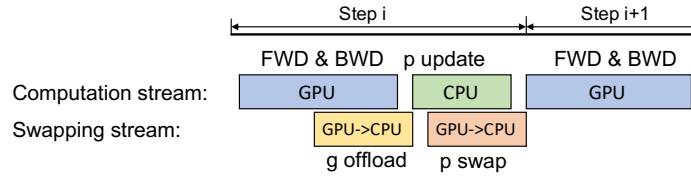


Figure 5.3: ZeRO-Offload training process on a single GPU.

aforementioned metrics, it is also unique; there can be no other strategy that can offer more memory savings than ZeRO-Offload without increasing the compute complexity on the CPU or incur additional GPU-CPU communication volume.

5.4 ZeRO-Offload Schedule

In this section, we discuss the concrete computation and communication schedule for implementing ZeRO-Offload on a single GPU system based on our offload strategy. We then show how we extend this schedule to work effectively on multi-GPU systems by combining our offload strategy with ZeRO data parallelism and model parallelism.

5.4.1 Single GPU Schedule

As discussed in Sec. 5.3, ZeRO-Offload partitions the data such that the fp16 parameters are stored in GPU while the fp16 gradients, and all the optimizer states such as fp32 momentum, variance and parameters are stored in CPU.

During the training, we begin by computing the loss via the forward propagation. Since the fp16 parameters are already presented on GPU, no CPU communication is required for this part of the computation. During the backward propagation on the loss, the gradient for different parameters are computed at different point in the backward schedule. ZeRO-Offload can transfer these gradients for each parameter individually or in small groups to the CPU memory immediately after they are computed. Therefore, only a small amount of memory is required to temporarily hold the gradients on the GPU memory before they are transferred to CPU memory. Furthermore, each gradient transfer can be overlapped with the backpropagation on the remainder of the backward graph, allowing ZeRO-Offload to hide a significant portion of the communication cost.

After the backward propagation, ZeRO-Offload updates the fp32 parameters and the remaining optimizer states (such as momentum and variance) directly on CPU, and copies the updated fp32 parameters from the CPU memory to the fp16 parameters on the GPU memory. Figure 5.3

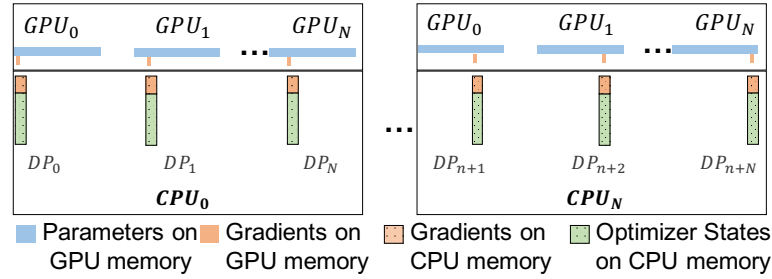


Figure 5.4: ZeRO-Offload data placement with multiple GPUs

shows the computation and communication in each step of ZeRO-Offload diagrammatically, and Figure 5.5 shows the concrete schedule as a pseudo-code.

5.4.2 Scaling to Multi-GPUs

ZeRO-Offload in its entirety is a symbiotic integration of ZeRO-Offload strategy described in Sec. 5.3 and ZeRO-powered data parallelism discussed in Sec. 5.2, which allows ZeRO-Offload to scale to hundreds of GPUs efficiently. ZeRO-Offload preserves the model state partitioning strategy of ZeRO Stage-2 (optimizer state and gradient partitioning), while offloading the partitioned gradients, optimizer states and the corresponding parameter updates to CPU.

The key benefit of doing this partitioning before offloading is that for systems with more than one GPU, each data parallel process is only responsible for updating a subset of the parameters. The aggregated communication volume from all the data parallel GPUs to CPU remains constant, and CPU resources are used in parallel to jointly compute a single weight update. As a result, the total CPU update time decreases with increased data parallelism, since the CPU compute resources increase

```

1  for_parallel rank in range(world_size):
2      initialize_layers()
3      for batch in dataset:
4          x = forward(batch)
5          compute_loss(x, batch).backward()
6          backward(x.grad)
7          step()
8
9  def _is_owner(i):
10     return True if rank owns i else False
11
12  def initialize_layers():
13     for i in range(num_layers):
14         l = layers[i]
15         allocate_on_gpu l.param_fp16
16         if _is_owner(i):
17             allocate_on_cpu l.param_fp32
18             allocate_on_cpu l.optim_states_fp32
19             allocate_on_cpu l.cpu_grad
20
21  def forward(x):
22     for i in range(num_layers):
23         x = layers[i].forward(x)
24     return x
25
26  def backward(dx):
27     for i in range(num_layers, 0, -1):
28         dx=layers[i].backward(dx)
29         reduce(layers[i].grad, dest_rank
30              = _owner_rank(i))
31         if _is_owner(i) l.cpu_grad.copy(l.grad)
32         else pass
33         del layers[i].grad
34
35  def step():
36     for i in range(num_layers):
37         l=layers[i]
38         if _is_owner(i):
39             update_in_cpu(l.optim_states_fp32,
40                          l.cpu_grad,
41                          l.param_fp32)
42             l.param_fp16.copy(l.param_fp32)
43             BROADCAST(l.param_fp16, src=_owner_rank(i))

```

Figure 5.5: Code representing ZeRO-Offload that combines unique optimal CPU offload strategy with ZeRO-powered data parallelism.

linearly with the increase in the number of compute nodes. This allows ZeRO-Offload to achieve very good scalability, as the overhead of communication across GPUs is offset by the reduction in the CPU optimizer step.

ZeRO-Offload partitions gradients and optimizer states among different GPUs, and each GPU offloads the partition it owns to the CPU memory and keeps it there for the entire training. During the backward propagation, gradients are computed and averaged using reduce-scatter on the GPU, and each GPU only offloads the averaged gradients belonging to its partition to the CPU memory. Once the gradients are available on the CPU, optimizer state partitions are updated in parallel by each data parallel process directly on the CPU. After the update, parameter partitions are moved back to GPU followed by an all-gather operation on the GPU similar to ZeRO-2 to gather all the parameters. Figure 5.4 shows the data placement model parameters, gradients and optimizer states for ZeRO-Offload and the details of the ZeRO-Offload data parallel schedule is presented in Figure 5.5. The all gather operation described above is shown as a sequence of broadcast operations in the Figure.

Model Parallel training ZeRO-Offload can also work together with tensor-slicing based model parallelism (MP) frameworks such as Megatron-LM [195]. It does so by offloading the gradients, optimizer states and the optimizer computation corresponding to each MP process allowing ZeRO-Offload to train significantly larger models than possible than using model parallelism alone. Sec. 5.6 provides more details.

5.5 Optimized CPU Execution

We speedup the CPU execution time for the parameter updates with two optimizations. First, we implement a fast CPU Adam optimizer using high performance computing techniques offering significant speedup over state-of-art Pytorch implementation. Second, we develop a one-step delayed parameter update schedule that overlaps the CPU parameter update computation with the forward and backward computation on the GPU, hiding the CPU execution time when enabled.

5.5.1 Implementing the CPU Optimizer

We use three levels of parallelism for improving the performance of the CPU optimizer. 1) SIMD vector instruction [135] for fully exploiting the hardware parallelism supported on CPU architectures. 2) Loop unrolling [209], an effective technique for increasing instruction level parallelism that is crucial for better memory bandwidth utilization. 3) OMP multithreading for

effective utilization of multiple cores and threads on the CPU in parallel. Using these technique, we present a significantly faster implementation of Adam optimizer compared to state-of-art PyTorch implementation.

Mixed Precision Training with Adam ADAM is an optimization algorithm used for deep-learning training, which takes the loss gradients together with their first and second momentums to update the parameters. Therefore, in addition to the model parameters, ADAM requires two more matrices of the same size (M) saved during the training. In the mixed precision training mode, there are two versions of the parameters stored in memory: one in fp16 (parameter16) used for computing the activations in the forward pass (on GPU), and one master copy in fp32 (parameter32) which is updated by the optimizer (on CPU). The p16 is updated with the parameter32 through *float2half* casting, at each training step. Moreover, the momentum and variance of the gradients are saved in fp32 (on CPU), to prevent the precision loss for updating the parameters. Please refer to [99] for further detail on ADAM’s algorithm.

Algorithm 4: CPU-ADAM Optimizer

Input: $p32, g32, m32, v32, \beta_1, \beta_2, \alpha, step, eps$

Output: $p16, p32, m32, v32$

Parameters : $tile_width, simd_width, unroll_width$

```

1   $biascorrection1 \leftarrow -\alpha / (1 - \beta_1^{step})$ 
2   $biascorrection2 \leftarrow 1 / \sqrt{1 - \beta_2^{step}}$ 
3   $simd\_count \leftarrow sizeof(32) / simd\_width$ 
4  unroll omp parallel
5  for  $i$  in 1 to  $(simd\_count / unroll\_width)$  do
6  |   ...
7  |    $g_v, p_v, m_v, v_v = g32[i], p32[i], m32[i], v32[i]$ 
8  |    $m_v = FMA(g_v, (1 - \beta_1), \beta_1 * m_v)$ 
9  |    $v_v = FMA(g_v * g_v, (1 - \beta_2), \beta_2 * v_v)$ 
10 |    $g_v = FMA(\sqrt{v_v}, biascorrection2, eps)$ 
11 |    $g_v = m_m / g_v$ 
12 |    $p_v = FMA(g_v, biascorrection1, p_v)$ 
13 |    $p32[i], m32[i], v32[i] = p_v, m_v, v_v$ 
14 |   ...
15 |   IF  $(i == tile\_width)$  copy_to_gpu( $p16, p32$ )

```

Optimized Implementation Algorithm 4 elaborates the ADAM’s implementation detail using SIMD operations. As shown, the Adam function receives the optimizer parameters such as $\beta_1, \beta_2,$

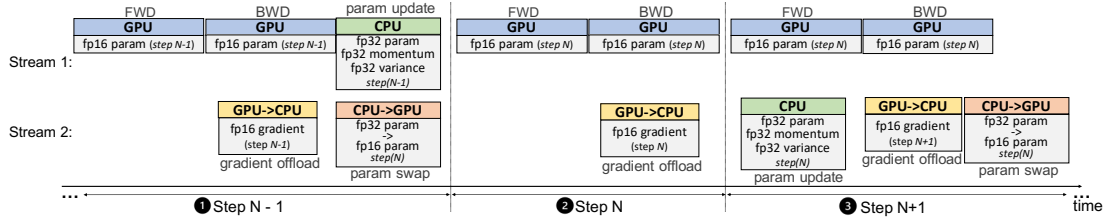


Figure 5.6: Delayed parameter update during the training process.

and α , and the gradient, momentum, variance and master copy of parameters (parameter32) as the input. We also use some parameters specific to the implementation, like the *simd_width* and *unroll_width*. The Adam optimizer sends back the updated variance, momentum, and parameter in both fp16 (to GPU) and fp32 (to CPU).

We firstly read the data, including parameter, gradient, momentum and variance, into the vector registers (line 7). Then, we use several fused multiply-add (*FMA*) vector operations to preform the main execution pipeline which is repeated by the unrolling width. Note that the rest of operations, such as multiply, division, and sqrt, also run in vector mode. For the best performance we use AVX512 simd instruction set and an *unroll_width* of 8 based on auto-tuning results.

In addition to the CPU-Adam optimizer, we implement the CPU-to-GPU fp16 parameter-copy in a tiled manner (line 15). We overlap the CPU and GPU execution by parallelizing the Adam computation and copying the parameters over to GPU. As we process Adam computation of the current tile of data on CPU, we write the parameters back to GPU for the previously processed tile. This way, we reduce the idle time of GPU to start the processing of the next training step.

5.5.2 One-Step Delayed Parameter Update

Despite using a highly optimized CPU optimizer, the CPU computation overhead can become a bottleneck during training with very small batch sizes, when the GPU computation time is not much larger than CPU compute. For such limited cases, we develop one-step delayed parameter update (DPU) that overlaps CPU and GPU compute to hide the CPU computation overhead by delaying the parameter update by a single step. We verify that DPU does not impact the final accuracy of training in the evaluation.

DPU training schedule Figure 5.6 shows the workflow of ZeRO-Offload training process with delayed parameter update. ❶ The first $N - 1$ steps, are trained without DPU to avoid destabilizing the training during the early stages where gradients change rapidly. ❷ On step N , we obtain the gradients from the GPU, but we skip the CPU optimizer step, and do not update

the fp16 parameters on the GPU either. ③ At step $N + 1$, we compute the parameter updates on the CPU using gradients from step N , while computing the forward and backward pass on the GPU in parallel using parameters updated at step $N - 1$. From this step onwards, the model at $(i + 1)^{th}$ step will be trained using the parameters updated with gradients from $(i - 1)^{th}$ step instead of parameters updated at i^{th} step, overlapping CPU compute with GPU compute.

Accuracy trade-off. Since DPU changes the semantics of the training, it is reasonable to ask if there is a trade-off between model accuracy and training efficiency. To answer this question, we evaluated DPU on multiple training workloads and found that DPU does not hurt convergence if we introduce DPU after a few dozen iterations instead of introducing it at the beginning. Our evaluation result in Sec. 5.6 shows that compared with training with ZeRO-Offload only, training with delayed parameter update achieves same model training accuracy with higher training throughput.

5.6 Evaluation

This section seeks to answer the following questions, in comparison to the state-of-the-art:

- (i) How does ZeRO-Offload scale the trainable model size compared to existing multi-billion parameter training solutions on a single GPU/DGX-2 node?
- (ii) What is the training throughput of ZeRO-Offload on single GPU/DGX-2 node?
- (iii) How does the throughput of ZeRO-Offload scale on up to 128 GPUs?
- (iv) What is the impact of our CPU-Adam and delay parameter update (DPU) on improving throughput, and does DPU change model convergence?

5.6.1 Evaluation Methodology

Testbed. For the evaluation of model scale and throughput, we conduct our experiments on a single DGX-2 node, whose details are shown in Table 5.2. For the evaluation of throughput scalability, we conduct experiments on 8 Nvidia DGX-2 nodes connected together with InfiniBand using a 648-port Mellanox MLNX-OS CS7500 switch.

Workloads. For the performance evaluation, we focus on evaluating GPT-2 [170] like Transformer based models [207]. We vary the hidden dimension and the number of Transformer blocks to obtain models with a different number of parameters. Note that scaling the depth alone is often not sufficient because it would make training more difficult [97]. Table 5.3 shows the configuration parameters used in our experiments.

Table 5.2: Hardware overview of experimental system.

DGX-2 node	
GPU	16 NVIDIA Tesla V100 Tensor Core GPUs
GPU Memory	32GB HBM2 on each GPU
CPU	2 Intel Xeon Platinum 8168 Processors
CPU Memory	1.5TB 2666MHz DDR4
CPU cache	L1, L2, and L3 are 32K, 1M, and 33M, respectively
PCIe	bidirectional 32 GBps PCIe

For convergence analysis, such as the delayed parameter update, we use GPT-2 [170] and BERT [43], both of which are commonly used as pre-trained language models and have demonstrated superior performance in many NLP tasks (e.g., natural language understanding and inference) than recurrent neural networks or convolutional neural networks. We use BERT-large, same as the one from [43], which has 24-layer, 1024-hidden, 16-heads, and 336M parameters. Similar to [172, 195], we fine-tune BERT on the Stanford Question Answering Dataset (SQuAD) [3], which is one of the most widely used reading comprehension benchmark [174]. Unless otherwise stated, we follow the same training procedure and hyperparameter settings as in [43, 170].

Baseline. We compare the effectiveness of ZeRO-Offload with state-of-arts multi-billion parameter training solutions:

- PyTorch DDP: This is the existing PyTorch Transformer implementation using Distributed Data Parallel [111].
- Megatron [195]: One of the current state-of-the-art multi-billion parameter model training solutions, which employs model parallelism to train up to 8.3B parameter models using 512 GPUs.
- SwapAdvisor [79]: SwapAdvisor explores a genetic algorithm to guide model-agnostic tensor swapping between GPU and CPU memory for GPU memory saving.
- L2L [164]: L2L enables training of deep Transformer networks by keeping one Transformer block at a time in GPU memory and only moves tensors in the upcoming Transformer block into GPU memory when needed.
- ZeRO-2 [172]: ZeRO extends data parallelism by eliminating memory redundancies across multiple GPUs, allowing to train models up to 170B parameters with high training throughput using 25 DGX-2 nodes. ZeRO-2 achieves the SOTA results for large model training and is a strong baseline.

Table 5.3: Model configuration in evaluation.

# params	batch size per GPU	MP setting in	# layer	hidden size
1, 2 billion	32	1	20, 40	2048
4 billion	32	1	64	2304
6, 8 billion	16	1	53, 72	3072
10,11 billion	10,8	1	50,55	4096
12, 13 billion	4	1	60, 65	4096
15 billion	8	2	78	4096
20,40,60 billion	8	2	25,50,75	8192
70 billion	8	8	69	9216

5.6.2 Experimental Results

Model Scale

As an important step toward democratizing large model training, in this part, we first test the largest trainable models on a single GPU as well as 16 GPUs in a single DGX-2 node.

Single GPU. The largest model can be trained using PyTorch DDP on a single GPU with 32GB memory is 1.4B, before running out of memory, as shown in figure 5.7. Both Megatron and ZeRO-2 do not increase the trainable model size on a single GPU in comparison to PyTorch, because they both utilize the aggregated GPU memory to fit larger models. In contrast, ZeRO-Offload enables 13B model training on a single GPU, which is more than 9X larger than using PyTorch, Megatron, and ZeRO-2. This is mainly because of ZeRO-Offload’s strategy for maximizing the memory savings on GPU by offloading expensive states such as optimizer states and the majority of gradients to CPU memory. The largest model can be trained with SwapAdvisor on a single GPU is 8B, which is 38% smaller than the model can be trained with ZeRO-Offload. SwapAdvisor relies on a black-box approach and uses a simulator to predict which tensors are more frequently used in order to keep them in GPU memory to maximize training throughput. The prediction can not be fully accurate, and therefore SwapAdvisor keeps more tensors in GPU memory than ZeRO-Offload does. On the other hand, L2L is able to train even larger models (e.g., 17B) on a single GPU by frequently moving weights from unused layers to CPU memory. However, the largest model size does not increase when training L2L with multiple GPUs, which is discussed next.

Multi-GPU in single DGX-2. We further perform model scale tests with 4 and 16 GPUs in a single DGX-2 node, respectively. As shown in Figure 5.7, the maximum trainable model size stays the same for PyTorch, L2L and SwapAdvisor, because all of them do not handle memory

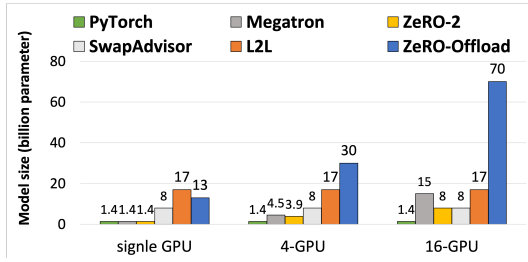


Figure 5.7: The size of the biggest model that can be trained on single GPU, 4 and 16 GPUs (one DGX-2 node).

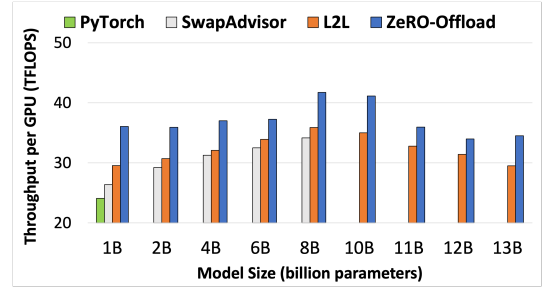


Figure 5.8: The training throughput with PyTorch, L2L, SwapAdvisor and ZeRO-Offload on a single GPU with a batch size of 512.

redundancies in data parallelism. As a result, their scalability is bounded by the model scale on a single GPU. Both Megatron and ZeRO-2 support large model training with more GPUs, but they cannot scale efficiently beyond 15B parameters, even with 16 GPUs. Megatron supports larger models than ZeRO-2, because ZeRO-2 still incurs memory redundancies on model weights. On the other hand, ZeRO-Offload easily enables training of up to 70B parameter models by partitioning and offloading optimizer states and gradients to CPU memory combined with model parallelism. Overall, ZeRO-Offload increases the model scale on a single DGX-2 node by 50X, 4.5X, 7.8X, and 4.2X than using PyTorch, Megatron, ZeRO-2, and L2L, respectively.

Training Throughput

Single GPU. Next, we compare the training throughput of SwapAdvisor, L2L and ZeRO-Offload, for models with billion-scale parameters, on a single GPU. We do not include Megatron and ZeRO-2 in this comparison, because both of them cannot train models bigger than 1.4B parameters due to OOM. We evaluate SwapAdvisor, L2L and ZeRO-Offload with the same training batch size (e.g., 512) and same micro-batch sizes (shown in table 5.3), with gradient accumulation enabled. We also disable delayed parameter update in this experiment so that the comparison is only from the system efficiency perspective. We evaluate the performance improvement and its impact on the convergence of delayed parameter update in Section 5.6.2.

Figure 5.8 shows that ZeRO-Offload outperforms SwapAdvisor by 23% (up to 37%) in training throughput. SwapAdvisor relies on online genetic algorithm to make tensor swapping decision, which takes hours to find an optimal tensor swapping solution in terms of maximizing the overlapping of computation and tensor swapping. Before getting the optimal tensor swapping solution, SwapAdvisor tries random tensor swapping solutions and hurts training performance.

Figure 5.8 shows that ZeRO-Offload outperforms L2L by 14% on average (up to 22%) in throughput (TFLOPS). The performance benefit of ZeRO-Offload comes from the following two aspects. First, ZeRO-Offload has a lower communication cost between CPU and GPU than L2L. For a model with M parameters, L2L requires $28M$ data communication volume between GPU and CPU, which is a sum of the weights, gradients, and optimizer states of each layer of the model. As analyzed in Sec. 5.4.1, the communication volume between CPU and GPU memory in ZeRO-Offload is $4M$, which is 7x smaller than L2L. The reduced communication volume significantly mitigates the bottleneck from CPU-GPU communication. Second, compared with L2L, the parameter update of ZeRO-Offload happens on CPU instead of GPU, but our optimized CPU-Adam implementation achieves a quite comparable parameter update performance than the PyTorch Adam implementation on GPU (evaluated in Sec. 5.6.2). Therefore, although the optimizer update on GPU in L2L is slightly faster than the optimizer update on CPU in ZeRO-Offload, the communication overhead introduced by L2L leads to an overall slower throughput than ZeRO-Offload.

Multi-GPU in single DGX-2. Next, we compare the training throughput of PyTorch, ZeRO-2, Megatron, ZeRO-Offload without model parallelism (w/o MP), and ZeRO-Offload with model parallelism (w/ MP) in one DGX-2 node. When using MP, we use a MP degree that gives the best performance for both baseline and ZeRO-Offload. We use a total batch size of 512 for all the experiments using a combination of micro-batch per GPU and gradient accumulation. To get the best performance for each configuration, we use the largest micro batch that it can support without OOM. We exclude L2L [202] in this test because its implementation does not support multi-GPU training.

Figure 5.9 shows the throughput per GPU results when training on multiple GPUs. We make the following observations:

- For 1B to 15B models, ZeRO-Offload achieves the highest throughput and has up to 1.33X, 1.11X, 1.64X higher speeds than PyTorch, ZeRO-2, and Megatron, respectively. By offloading all the optimizer states to CPU with low overhead, ZeRO-Offload can train with larger micro-batch sizes giving higher throughput.
- ZeRO-2 runs out of memory once the model size is beyond 8B due to lack of enough aggregated GPU memory to store the model states on 16 GPUs. Instead, ZeRO-Offload scales to 13B, without model parallelism because it offloads optimizer states and the majority of gradients to CPU memory.
- When combined with model parallelism, ZeRO-Offload enables training up to 70B parameter

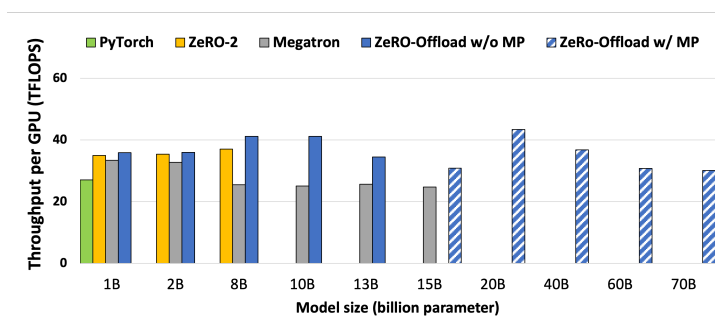


Figure 5.9: Training throughput with PyTorch, ZeRO-2, Megatron-LM, ZeRO-Offload without model parallelism and ZeRO-Offload with model parallelism.

models with more than 30 TFLOPS throughput per GPU. In contrast, Megatron supports only up to 15B parameter models before running out of memory, using just model parallelism.

- Compared ZeRO-Offload with ZeRO-2 and Megatron, ZeRO-Offload outperforms ZeRO-2 and Megatron in throughput for 1–8B and 1–13B parameter models, respectively. ZeRO-Offload is faster than Megatron, because it eliminates frequent communication between different GPUs and can train with larger micro batch sizes. ZeRO-Offload outperforms ZeRO-2 also due to larger micro batch sizes.

Throughput Scalability

We compare the throughput scalability of ZeRO-2 and ZeRO-Offload³ on up to 128 GPUs in Figure 5.10 and make the following key observations: First, ZeRO-Offload achieves near perfect linear speedup in terms of aggregated throughput (green line) running at over 30 TFlops per GPU (blue bars). Second, from 1 to 16 GPUs, while ZeRO-2 runs out of memory, ZeRO-Offload can effectively train the model, turning the model training from infeasible to feasible. Third, with 32 GPUs, ZeRO-Offload slightly outperforms ZeRO-2 in throughput. The improvement comes from additional memory savings on GPU from ZeRO-Offload, which allows training the model with larger batch sizes that lead to increased GPU computation efficiency. Fourth, with more GPUs (such as 64 and 128), ZeRO-2 starts to outperform ZeRO-Offload, because both can now run similar batch sizes, achieving similar computation efficiency, whereas ZeRO-2 does not suffer from the additional overhead of CPU-GPU communication. In summary, ZeRO-Offload complements ZeRO-2 and enables large model training from a single device to thousands of devices with good computation efficiency.

³We do not include comparison against Megatron because it consistently performs worse than ZeRO-Offload, as shown in Figure 5.9. Given the communication overhead added by model parallelism, scaling out Megatron training

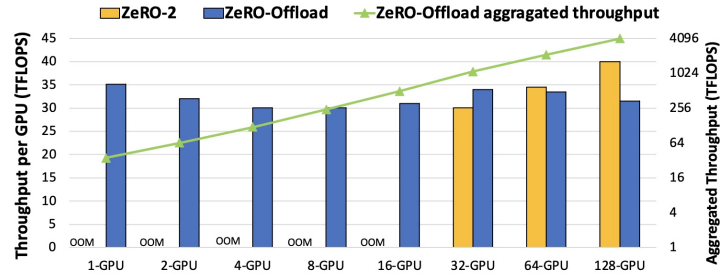


Figure 5.10: Comparison of training throughput between ZeRO-Offload and ZeRO-2 using 1–128 GPUs for a 10B parameter GPT2.

Optimized CPU Execution

A. CPU-Adam efficiency. In this part, we evaluate our Adam implementation against the PyTorch Adam on CPU. Table 5.4 shows the optimizer execution time of the two implementations for model parameters from 1 to 10 billion. Compared to PyTorch (PT-CPU), CPU-Adam reduces the execution time by over 5X for all the configurations and 6.4X for the case with 1B parameters. The CPU-Adam optimizer achieves high speedups by exploiting the instruction-level parallelism, thread-level parallelism, and the tile-based data copy scheme (as shown in line 15 of Algorithm 4). Meanwhile, although CPU-Adam has a slower speed than the PyTorch Adam implementation on GPU (PT-GPU), the performance gap is not very huge, and the CPU computation is not a bottleneck of the training throughput.

B. One-step Delayed parameter update (DPU). Figure 5.11 shows the comparison of the training throughput of GPT-2 with and without DPU. As shown, with DPU enabled, the training achieves 1.12–1.59, updated times higher throughput than without it, for a wide range of model sizes for a small micro batch size of 8. This is expected because DPU allows the optimizer updates to overlap with the next forward computation such that the GPU does not have to be slowed down by the CPU computation and CPU-GPU communication. But, what about accuracy?

Convergence impact We study the convergence impact of DPU on both GPT-2 and BERT. Figure 5.12 shows the pre-training loss curves over 100K training iterations using PyTorch (unmodified GPT-2), and Figure 5.13 shows the loss curves of fine-tuning Bert-large model on SQuAD using ZeRO-Offload without DPU, and ZeRO-Offload with DPU. In both cases, DPU is enabled after 40 iterations allowing the training to stabilize in its early stage before introducing DPU.

We observe that the training curves of the unmodified GPT-2 and ZeRO-Offload w/o DPU are can not achieve higher throughput than ZeRO-Offload even with linear scalability.

Table 5.4: Adam latency (s) for PyTorch (PT) and CPU-Adam.

#Parameter	CPU-Adam	PT-CPU	PT-GPU (L2L)
1 billion	0.22	1.39	0.10
2 billion	0.51	2.75	0.26
4 billion	1.03	5.71	0.64
8 billion	2.41	11.93	0.87
10 billion	2.57	14.76	1.00

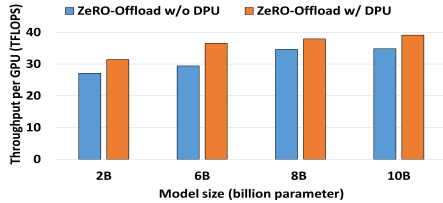


Figure 5.11: The training throughput is compared for w/o DPU and w/ DPU to GPT-2. Batch size is set to 8.

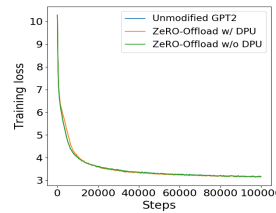


Figure 5.12: The training loss curve of unmodified GPT-2, ZeRO-Offload w/o DPU and ZeRO-Offload with DPU.

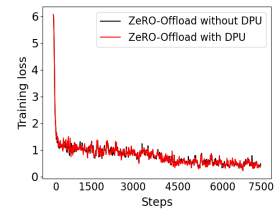


Figure 5.13: The fine-tuning loss curve of BERT, ZeRO-Offload w/o DPU and ZeRO-Offload with DPU.

exactly overlapped, because ZeRO-Offload w/o DPU performs only system optimizations and does not alter training dynamics. On the other hand, the training curve from ZeRO-Offload with DPU converges slightly slower at the very beginning of the training (e.g., barely can be seen at 2K-5K iterations) and quickly catches up after 5K iterations. For the remaining of the training, the training loss matches the original training until the model converges.

For Bert-Large fine-tuning, we can see that although the training losses are not exactly the same, they converge in the same trend and are largely overlapped. Without changing any hyperparameters, ZeRO-Offload + DPU achieves the same final F1 score (92.8) as the baseline. From these results on both GPT-2 pretraining, and Bert-Large fine-tuning, we empirically verify that DPU is an effective technique to improve the training throughput of ZeRO-Offload without hurting model convergence and accuracy. The 1-step staleness introduced by DPU is well tolerated by the iterative training process once the model has passed the initial training phase.

Performance Breakdown and Analysis

To better understand the performance benefit from offload strategies and optimization techniques in ZeRO-Offload, we evaluate the training throughput of PyTorch, ZeRO-Offload with PT-CPU, ZeRO-Offload with CPU-Adam (refer as ZeRO-Offload), and ZeRO-Offload with DPU. We

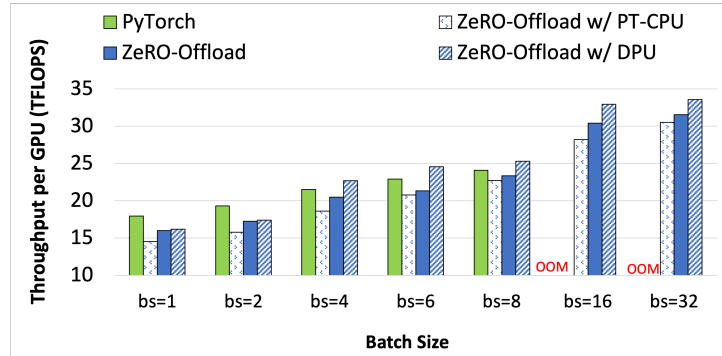


Figure 5.14: Comparison of training throughput with enabling offload strategies and optimization techniques step-by-step in ZeRO-Offload.

perform the evaluation with various batch sizes with 1-billion GPT-2 model on a single GPU. Figure 5.14 shows the result.

From batch size 1 to 8, PyTorch outperforms ZeRO-Offload with PT-CPU by 16% on average. This is because when the model can fit on GPU memory, PyTorch does not incur any communication overhead. Meanwhile, PyTorch adopts PyTorch GPU Adam (PT-GPU) for optimizer computation on GPU. To reduce the performance loss because of communication and optimizer computation on CPU, ZeRO-Offload optimizes execution on CPU. (1) By optimizing CPU optimizer, ZeRO-Offload implements CPU-Adam and improves the performance by up to 10% compared with using offload strategy only (i.e., ZeRO-Offload with PT-CPU). (2) PyTorch outperforms ZeRO-Offload by 8% on average when the model can fit on GPU memory. As shown in table 5.4, the performance gap between CPU-Adam and PT-GPU is not very large. Therefore, the performance degradation from PyTorch to ZeRO-Offload in Figure 5.14 are mainly coming from tensor migration overhead between GPU and CPU memory. (3) ZeRO-Offload further introduces one-step delayed parameter update, which overlaps computation on CPU with computation on GPU and improves performance by 7% compared with using ZeRO-Offload without DPU. In summary, leveraging optimized CPU execution, ZeRO-Offload has similar performance as PyTorch when ZeRO-Offload and PyTorch training with the same batch size on GPU.

As the batch size increases, out-of-memory on GPU memory happens in training with PyTorch. The training throughput increases in ZeRO-Offload as the batch size increasing. With unique optimal offload strategy, ZeRO-Offload outperforms PyTorch by 39% for the maximum training throughput that can be achieved on a single GPU with 1-billion model.

5.7 Conclusions

We presented ZeRO-Offload, a powerful GPU-CPU hybrid DL training technology with high compute efficiency and near linear throughput scalability, that can allow data scientists to train models with multi-billion parameter models even on a single GPU, without requiring any model refactoring. We open-sourced ZeRO-Offload as part of the DeepSpeed library (www.deepspeed.ai) with the hope to democratize large model training, allowing data scientists everywhere to harness the potential of truly massive DL models.

Chapter 6

Enabling Large Dynamic Neural Network Training with Learning-based Memory Management

Dynamic neural network (DyNN) enables high computational efficiency and strong representation capability, and hence has been applied to many important problems. However, DyNN faces a memory capacity problem because of increasing model size or limited GPU memory capacity. Managing tensors to save GPU memory is challenging, because of the dynamic structure of DyNN. This paper presents DyNN-Offload, a memory management system to train DyNN. Uniquely, DyNN-Offload uses a learned approach (using a neural network) to increase predicability of tensor accesses to facilitate memory management.

The key of DyNN-Offload is to enable fast inference while providing high prediction accuracy of the learned model. DyNN-Offload reduces input feature space and model complexity based on a new representation of DyNN; DyNN-Offload converts the hard problem of making prediction for individual tensors or operators into a simpler problem of making prediction for a group of operators. To hide communication latency incurred by tensor migration, DyNN-Offload learns knowledge on tensor migration from static neural networks. DyNN-Offload outperforms state-of-the-art solutions by 2%-50% in terms of training time with the same GPU memory capacity and enables 8x larger model training without out of memory. DyNN-Offload demonstrates the possibility of using a learned approach to remove dynamism and address complicated problems on performance optimization and analysis.

6.1 Introduction

Deep learning (DL) is embracing dynamic neural neural (NN) architectures where the NN structure changes for each data sample. Such dynamic neural networks (DyNN) are different from the traditional static NN where a network architecture (i.e., a dataflow graph) is defined using symbolic expressions, once before beginning execution. In a dataflow graph, computation functions in NN are associated with nodes, and input and output of the computation map to edges. With a static NN, the dataflow graph is fixed for all data. In contrast, with DyNN, the DL model may select its model components (e.g., layers [80], channel [117] or sub-networks [193]) conditional on input samples, and change the structure and parameters in the dataflow graph accordingly. The DyNN has shown high computational efficiency computing over sequences of variable lengths [199], trees [200], and graphs [114]. It also shows strong representation capability and high adaptiveness to achieve a desired trade-off between accuracy and efficiency on the fly [70]. As a result, DyNNs have been applied to many important problems, such as speech recognition [229], DL translation [25, 199], and question answering [19].

Problems. DyNN, as many other NN, faces a memory-capacity problem [158, 178, 179, 240]. As the increase of model size brings substantial quality gains, we have seen the emergence of large models recently. For example, DyNN var-Bert with 48 transformer-layers consumes more than 20 GB memory, well beyond the memory capacity of an edge device (e.g., NVIDIA Jetson TX2 GPU which has 8 GB memory) or a desktop GPU (e.g., NVIDIA RTX 6000 which has 23 GB memory) where DyNNs are often deployed. Hence accessibility to large model training is severely limited by GPU memory capacity. Distributed parallel training technologies such as pipeline model parallelism [72, 81, 137] and tensor model parallelism [192] go beyond the memory boundaries of single GPU devices by splitting the model states across multiple GPU devices, which enables massive models that would otherwise not fit into a single GPU memory. However, these techniques require enough GPU devices to provide large aggregated GPU memory to save all the model states required for training, which is not affordable by many use scenarios [173, 179].

Exploiting CPU memory to save GPU memory for large model training by tensors offloading has been explored recently [77, 79, 158, 164, 178, 179, 182], but has difficulty to be applied to DyNN. In particular, using the heterogeneous memory (CPU memory plus GPU memory), one wants to minimize communication volume or hide communication overhead between CPU and GPU, while maximizing memory savings on GPU. To achieve the above goal, most of existing efforts rely on profiling-guided optimization (PGO) to record tensor access orders and guide tensor

migration between CPU and GPU. The success of PGO is based on a fundamental assumption: the NN model must be invariant, i.e., using a static computation graph where tensor dimensions are fixed, data flow and control flow are fixed, and there are no complex data structures (such as graphs and trees) in the dataflow graph. For such a model, profiling a few iterations of the training process is enough to guide tensor allocation and migration in the whole training process.

However, the above assumption for PGO is broken for DyNN due to the model’s dynamic nature. Depending on the input, the DyNN selectively activates model components, which introduces irregular memory accesses and invalidates profiling results across training iterations. As a result, the communication between CPU and GPU is largely exposed to the critical path, causing training throughput loss. Our evaluation shows that using L2L, a state-of-the-art, industry-quality framework to train large NN models using PGO and tensor offloading, to train Tree-CNN [185], 85% of tensor migration (in terms of tensor size) is exposed to the critical path and there is 35% loss in training throughput, compared with the case of no tensor migration cost. Hence, we lack a cost effective while high-performance solution to train large DyNN.

In this paper, we introduce a memory (tensor) management system, *DyNN-Offload*, for training large DyNN. Its design is based on a new approach to guide tensor migration between CPU and GPU for GPU memory saving. In particular, we explore the extent to which a *learned model*, such as an NN, can be used to increase predictability of tensor accesses during the training process of large DyNN. Based on the learned model, we are able to store tensors on CPU memory, but prefetch them to GPU memory to hide communication cost.

Major insights. The key to our approach is *learning*, i.e., using the learned knowledge proactively gained from other input problems and DyNNs, instead of using PGO which lacks flexibility to handle dynamism. Our approach is driven by the following two insights.

- The training process of NN (no matter static or dynamic) has learnable patterns. For example, it is common to have linear computation followed by nonlinear computation (e.g., ReLU activation function).
- In essence, the dynamic structure of DyNN is built upon a series of decision-making processes (i.e., control flows) to activate model components, which in turn impacts the access orders of tensors. The input sample to DyNN provides indications on how such decision-making processes happen. The machine learning opens up the opportunity to learn a model that enables automatic synthesis of the decision-making processes.

Research challenges. We face multiple challenges when materializing the learning-based memory management.

The first challenge is how to avoid performance impact of the learned model on training time. The inference of the learned model introduces performance overhead. A training iteration of a DyNN (e.g., Tree-LSTM with $N \times 1024 \times 1024$ parameters, where N depends on the input), consuming one training sample, can take as small as $\sim 100 \mu s$. The inference time of the learned model should be less than $\sim 10 \mu s$, a $< 10\%$ overhead per iteration. Fast inference depends on input preprocessing and feature representation. The input features must be minimized to include only those that matter.

The second challenge is how to make the inference highly usable. This means the inference results should be useful to decide when to prefetch tensors from CPU memory to GPU memory instead of on-demand fetching (e.g., as in unified memory [146]). Tensor prefetching should be overlapped with GPU computation as much as possible, in order to remove tensor migration overhead from the critical path.

We could build a learned model to predict exact execution order of operators, such that the exact prefetching time can be planned as with a static NN and PGO-guided tensor migration. However, this method requires rich output from the learned model and imposes high requirement on prediction accuracy, which leads to heaviness in the learned model.

The third challenge is how to reconcile the tension between prediction accuracy of the learned model and tensor migration overhead. The effectiveness of the inference results relies on the accuracy of control-flow prediction. A mis-prediction causes missing tensors in GPU memory when needed by an operator. As a result, tensor fetching has to happen on demand, losing training throughput. Accuracy can be improved by increasing the depth of the layers or neural complexity, but at the cost of long inference time. Hence, there is a fundamental tension between prediction accuracy and migration overhead.

DyNN-Offload. The design of the learned model in DyNN-Offload is centered around how to enable model lightness while providing high prediction accuracy. This is achieved based on two observations: (1) Operators in machine learning, although being rich in interfaces and algorithms, can be identified by a combination of six pervasive and expressive memory access patterns. (2) Tensors typically migrate in batches in order to fully utilize interconnect bandwidth. Among those tensors that migrate together, there is no need of predicting exact execution order of those operators that will work on the migrated tensors. This observation relaxes the requirement of using information of fine-grained execution order to plan tensor migration, which is rooted in the existing PGO-based solution for static NN [164, 178, 182, 219].

Based on the observation (1), the input features and output of the learned model are able

to use a compact representation based on counting of six patterns to represent the architecture of the DyNN and indicate execution order of operators. This compact representation leads to a reduction of input feature space and hence a simpler learned model, which cannot be achieved by using alternative representations (e.g., using operator types as input features). Based on the observation (2), the learned model implicitly partitions the DyNN with resolved dynamism into multiple execution blocks, and only predicts the execution order of those blocks, which leads to an easy prediction task, and hence lighter model and higher prediction accuracy. The above techniques address the first challenge on the performance overhead of the learned model.

To address the second challenge on planning tensor prefetching, DyNN-Offload resorts to learn how to hide tensor migration through the training of the learned model. This is achieved by transforming DyNNs into the static ones and then using an existing PGO solution to decide execution blocks. Such transformation allows DyNN-Offload to create training samples with the knowledge of optimal model partition.

To address the third challenge on reconciling the tension between prediction accuracy and migration overhead, DyNN-Offload does not aim to maximize prediction accuracy. Instead, DyNN-Offload amortizes the mis-prediction penalty for an input sample by passing the mis-prediction knowledge to predict control flows for the similar input samples. This avoids repeated mis-prediction.

Results. Our evaluation shows that DyNN-Offload supports a variety of DyNN models (CNN-based, LSTM-based, and transformer-based) and works on real production datasets without requiring the user to refactor DyNN models. Given the same GPU memory capacity for DyNN, in terms of training time, DyNN-Offload outperforms existing solutions (DTR [100] and unified memory) by 2%-50%, and outperforms industrial approaches ZeRO-Offload [179] and L2L [164] that *work only for static NN* and heavily relies on workload characterization and machine learning domain knowledge by 11%-34%. The learned model in DyNN-Offload causes only 12 μ s inference overhead for each training sample of DyNN and achieves accuracy of 82% to resolve dynamism (i.e., control flows) in DyNN. With DyNN-Offload, we show that we are able to train 8x larger DyNN than without DyNN-Offload.

Key takeaway. The idea of using a learned approach to save memory for DyNN can have far reaching implications for future performance optimization and analysis of machine learning workloads. The machine learning is becoming increasingly complicated in terms of model topology, composability and expressiveness, which creates challenges for resource allocation and scheduling [113, 159, 167]. Using a learned approach, DyNN-Offload makes it possible, *for*

the first time, to provide a lightweight, efficient, and live solution for machine learning with rich dynamism.

6.2 Background

6.2.1 Dynamic Neural Networks

Static NN applies fixed-structured operations to all input samples. For example, convolutional NNs apply fixed network architecture to fixed-sized images, and are able to capture the spatial invariance common in computer vision. However, besides images, many forms of data (e.g., sequences of variable lengths and graphs) are structurally complex, and cannot be captured by fixed-structured NNs. Dynamic neural networks can adapt their structures or parameters to the input sample. Such dynamism is able to reflect the complex structures of input data, hence leading to high execution efficiency and accuracy. Such dynamism is often controlled by confidence-based criteria and gating functions, which are implemented by using control flows.

Figure 6.1 gives an example of DyNN. This example is a constituency parsing problem in natural language processing (NLP) that determines the grammar type of internal nodes in DyNN. The structure of DyNN varies depending on the content of the sentence itself. Figure 6.1.b shows an example where each node maps to a fully-connected NN (FC) (shown in Figure 6.1.a). This example generates representations for the input sentence by traversing the parse tree bottom-up and combining the representations of each sub-tree using the DyNN (named Tree-FC). Each node takes a variable number of inputs (Line 1 in Figure 6.1.a) and returns to the parent node a vector representing the parsing semantics up to that node. We use this example throughout the paper.

As shown in this example, the same FC NN is constant in shape and repeated at each node in the DyNN. The node may have control flows to determine the dynamic structure of DyNN (see Line 9 in Figure 6.1.a as an example).

6.2.2 Breaking Memory Capacity Wall

As state-of-the-art deep learning models continue to grow, training them within the capacity of GPU memory becomes increasingly challenging. Such a memory capacity wall limits ability to explore training techniques and memory-intensive model architectures. There are several solutions to reduce memory consumption and address this problem, such as using low-precision tensors [236], distributed training [137, 138, 173, 195], tensor redundancy removal [172], tensor migration on heterogeneous memory [77, 79, 158, 164, 178, 179, 182], and tensor

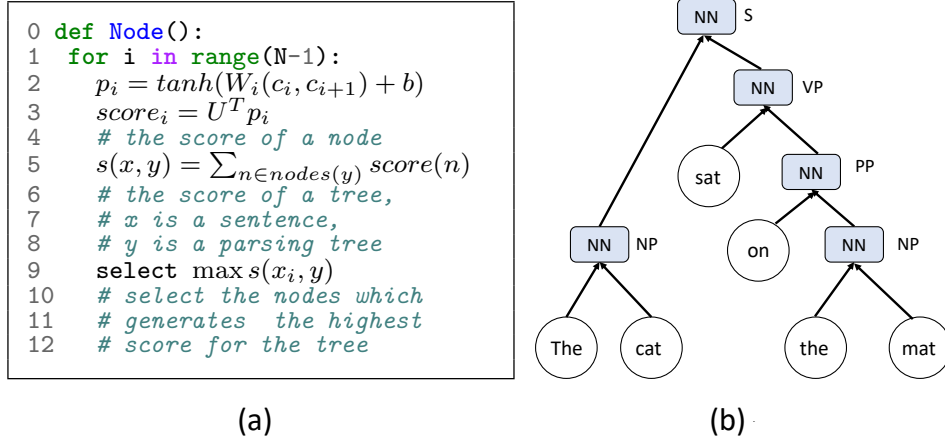


Figure 6.1: An DyNN example.(a) Implementation of each tree node in DyNN (b) The Tree-FC network where S, PP, NP and VP stand for sentence, preposition phrase, noun phrase, and verb phrase respectively.

Table 6.1: Distribution of Jaccard distance value for all training samples.

	[0,0.2]	(0.2,0.4]	(0.4,0.6]	(0.6, 0.8]	(0.8, 1]
Percentage of training samples	5%	28%	25%	40%	2%

rematerialization [38, 55, 88, 100, 196]. Among them, tensor migration and rematerialization are attractive, because they do not have risks of losing training convergence, do not change NN models, and are cost-effective (i.e., no need of extra GPU). However, when applying them to DyNN, they cannot work well.

Tensor migration highly relies on the workload predictability to decide when tensor migration should happen. For static NN, such predictability is provided by PGO based on the assumption that the workload characteristics (including execution time and tensor accesses) is invariant across training samples, which is not held for DyNN.

To quantify the unpredictability of DyNN, we use Tree-LSTM [200] with 6,000 training samples as an example. For each training sample, we build a binary vector and each element of the vector indicates if a specific control flow is taken or not. We use the first training sample as the baseline, and use the Jaccard distance [224] to quantify the execution similarity between the baseline and any other training sample. For a given training sample, the analysis result is a value falling into $[0, 1]$, with “1” indicating the training sample and the baseline take completely different control flows and “0” indicating opposite. A larger Jaccard distance value indicates lower similarity. Table 6.1 shows the results, which shows a wide divergence of execution of the dataflow graph across training samples. This divergence fails the traditional PGO-guided tensor migration, evidenced in our evaluation (Section 6.6.2).

Tensor rematerialization frees some tensors (particularly activations) from GPU memory but recomputes them on demand. Tensor rematerialization must use checkpointing to store some tensors in GPU, in order to replay the parent operations (a part of the forward pass) to reproduce the freed tensors. Tensor rematerialization can work for both static [38, 88, 102] and dynamic [100, 233] neural networks.

However, tensor rematerialization has fundamental limitations. (1) Rematerialization can be recursive: if the arguments to an freed tensor’s parent operation are freed too, then those arguments must first be rematerialized. There is no theoretical bound on depth of the recursiveness, leading to potential large loss in training throughput. (2) Some tensors (e.g., constant tensors and weights) cannot be rematerialized, leading to a tighter bound on memory saving (compared with using tensor migration techniques). Our evaluation shows the inferior performance of using tensor rematerialization than using the learned-model guided tensor migration (Section 6.6.2).

6.2.3 Using Machine Learning to Guide Tensor Migration

Before we explored machine learning models such as neural networks to guide tensor migration for DyNN, we asked whether simple heuristics would be accurate enough to resolve dynamism. For example, for a DyNN used in NLP and taking a sentence as input, one might measure the ratio of the number of verbs to the number of nouns in the input sentence, and assume that a larger ratio implies a higher possibility of taking a branch in the dataflow graph. However, we did not find a high correlation between the ratio and the decision of taking the branch, using Spearman’s correlation or Pearson’s correlation. Also, the heuristic is difficult to be generalized: different DyNNs or different prediction tasks require different heuristics. Hence, we decided to try machine learning models. Recent research on distributed and operating systems successfully employed machine learning for scheduling and resource allocation [48, 71, 167, 194, 230, 239]. A similar exploration aiming to address the memory capacity problem can lead to a powerful result, as we show in this paper.

6.3 Overview

We give an overview of DyNN-Offload.

Usage scenario. DyNN-Offload is used to make the decision on how input and output tensors of operators should be placed on CPU and GPU memories, in order to save GPU memory as much as possible without losing training throughput and accuracy. Before an input sample is fed

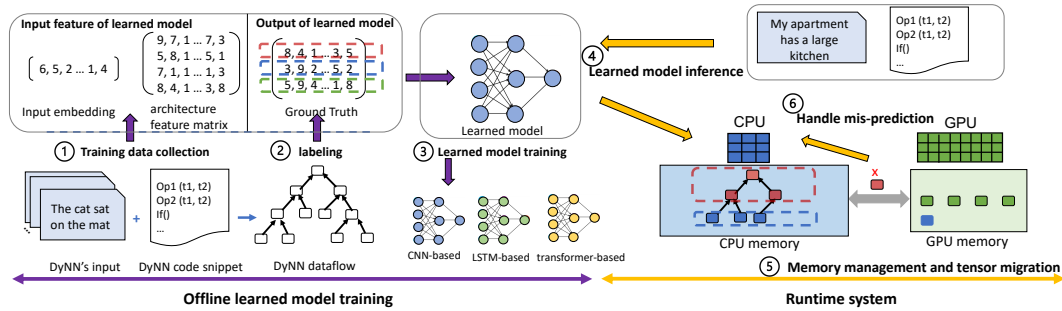


Figure 6.2: The overview of DyNN-Offload.

into the DyNN for training, an NN model (i.e., the learned model) in DyNN-Offload is used to quickly resolve the control flows and indicate when the tensor migration should happen. Based on the prediction of the learned model, the runtime system in DyNN-Offload controls tensor migration. To use DyNN-Offload, the user does not need to make any change to DyNN.

Overall architecture. Figure 6.2 shows the overall architecture of DyNN-Offload, consisting of three main components.

(1) The learned model. The center of DyNN-Offload is a light NN. The model’s input features are the input sample to the DyNN and the information about the DyNN architecture collected through static analysis on the DyNN model script. The model’s output indicates how operators will be executed at the granularity of execution blocks. Each execution block includes a group of operators. The learned model indicates how the computation graph of the DyNN is partitioned into execution blocks, such that at the beginning of an execution block i , the tensor migration for the next execution block is triggered to hide migration cost.

The input features and output of the learned model use a program idiom-based representation to identify operators. We use six idioms defined in terms of memory access patterns, and each operator can be easily characterized with a combination of the six idioms. Using the idiom-based representation significantly reduces the complexity of input features and output, compared with alternative solutions (e.g., using operator type), which leads to the lightness of the learned model.

(2) Runtime system. DyNN-Offload manages GPU memory for tensor migration and training based on a double-buffering mechanism. Based on the learned model, the tensors are able to be prefetched from CPU memory to GPU memory at the beginning of each execution block. DyNN-Offload also handles the mis-prediction of the learned model by fetching tensors on demand and recording the input sample to improve the accuracy of the learned model and avoid mis-prediction for other training samples.

(3) Training system for the learned model. To generate training samples to train the learned

model, DyNN-Offload feeds a number of DyNN’s input samples to different types of DyNN models, and records the execution trace of DyNN for tensor profiling. Then DyNN-Offload partitions the resolved dataflow graph for each DyNN’s input sample into execution blocks to maximize overlap between tensor migration and training computation. The information for those execution blocks plus DyNN’s input samples and architectures become training samples for the learned model.

6.4 Design

In this section, we describe our solution. To the best of our knowledge, DyNN-Offload is the first memory management system that enables large DyNN model training with a memory capacity constraint in a fast, accurate, and live fashion. The key to our design is the “lightness” of the NN model that DyNN-Offload employs. This section presents our design and the principal intuitions about how we get there. We will explain the three components of DyNN-Offload, from the learned model, (including input features (Section 6.4.1), output (Section 6.4.2), and model architecture (Section 6.4.3)), the runtime design (Section 6.4.5), to model training (Section 6.4.4).

6.4.1 Design of Input Features

The learned model takes the following information as input: (1) the input of the DyNN; and (2) the static architecture of the DyNN model. The static architecture is collected by static analysis on the DyNN modeling code. The static architecture includes *all* model components in the DyNN whose execution is determined by the control flows. The static architecture is different from the dynamic architecture which is input-dependent. We include the static architecture as input features, because the learned model is able to be independent of mode architecture and hence more general. We discuss how to represent the static architecture as input features of the learned model in this section.

Design Goals

When determining input features to represent the static architecture, we must reach the two goals: (1) We cannot have too many features, because that leads to a large learned model, causing large runtime overhead, and (2) The features should be informative to represent tensor accesses in operators.

We could use operator names (represented as numerical values) as the input features. In

particular, treating the computation in the DyNN as a sequence of operators, we employ a vector and each element of the vector represents an operator in the sequence. The operator names (or operator types), indicating how tensors are accessed, are informative. However, there are three problems with this solution. (1) There are a large number of operator names, leading to a large feature space. For example, in Pytorch, there are over three hundreds operator names. Using an operator name-based vector as the input increases the complexity of the learned model. (2) Some NNs have user-defined operators. Using the operator name as the input features lacks generality to handle a variety of NNs. (3) Some operators are the variant of the same operator (e.g., ADAM optimizer) but with different names. These operators access the same tensors and have the same functionality. There are no need to distinguish them in input features.

Idiom-based Representation

We introduce a new model representation as the input features to meet the above goals.

Idiom-based representation for operators. Each operator is characterized with six common idioms. An idiom is a computation pattern. Using this idiom-based representation is based on our observation that the six idioms have wide coverage of computation in machine learning operators. It has been shown that these six idioms are commonly found in numerical computation [35]. We describe these idioms using the following examples, where A , B , and C are two-dimensional vectors, i and j are indexes, a is a scalar tensor, and operators step through tensors by enumerating i and j .

- Transpose: $A_{ij} = B_{ji}$
- Gather: $A_{ij} = B_{C_{ii}C_{jj}}$
- Scatter: $B_{C_{ij}} = A_{ij}$
- Reduction: $a = a + A_{ij}$
- Stream: $A_{ij} = A_{ij} + B_{ij}$
- Stencil: $A_{ij} = A_{(i-1)j} + A_{(i+1)j}$

The six idioms are pervasive and expressive. We can find one or more of them (with possible repetitiveness) in the implementation of any operator (including the user-defined one). We build a signature for each operator based on the idioms. In particular, for an operator, we introduce a six-element vector where each element counts the number of occurrences of an idiom in the operator. To establish distinctive identification for each operator, we append operator input information to the six-element vector. An operator can have multiple input arguments, each of which has up to three dimensions. The operator input information is captured with a three-element vector

where each element is the accumulated dimension size of a dimension in all input arguments of the operator. Hence, each operator is represented with a nine-element vector.

Idiom-based representation for DyNN. Given a static architecture of a DyNN, we aggregate the nine-element vectors for all operators in each DyNN node into a matrix (named *architecture feature matrix* or *AFM*). In AFM, each row corresponds to one operator. The row order in AFM corresponds to the operator order in each DyNN node. In the case of a unresolved control flow, the operators in multiple branches are placed into AFM following the program order in the model script. If an operator occurs multiple times in the node, each occurrence has a row in AFM. Besides having operator representations, AFM has rows corresponding to the control flows in the node. Such a row has dummy values (all “0”s) and the row indexes in AFM correspond to the control statement locations in the static architecture. Hence, AFM represents the architecture of the DyNN and is used as an input feature.

The design of AFM pays great attentions to reduce the complexity of the learned model from the following two perspectives. *First*, the operators with the same tensor accesses and similar functionality are intentionally not distinguished to reduce the parameters of the learned model. For example, the activation functions, `ReLU` and `Sigmoid`, use the same idioms (i.e., `stream`) and tensor shapes. Those operators are not distinguishable in AFM. However, this does not impact the prediction accuracy of the learned model, because from the perspective of tensor usage, those operators do not have difference. *Second*, the detailed tensor information (such as tensor association with operators) is not explicitly expressed to reduce the row dimension in AFM. Instead, the tensor information is implicitly encoded in the operator information.

An example of AFM. Figure 6.3 shows the AFM for the Tree-FC in Figure 6.1. Each node in the Tree-FC has four operators, represented as the first four rows in AFM where the first six elements in each row correspond to the numbers of transpose, gather, scatter, reduction, stream, and stencil respectively. The last row in FAM corresponds to the control flow (Line 9 in Figure 6.1.a). The sequence of the operators and control flow follows the program order shown in Figure 6.1.a.

6.4.2 Output of the Learned Model

The learned model predicts how operators will be executed, by which the model implicitly indicates how the control flows (if there is any) in the DyNN will be resolved. The operators in the output of the learned model use the six idioms as a part of their representations, as in the input features, in order to reduce the complexity of the learned model. The learned model also decides

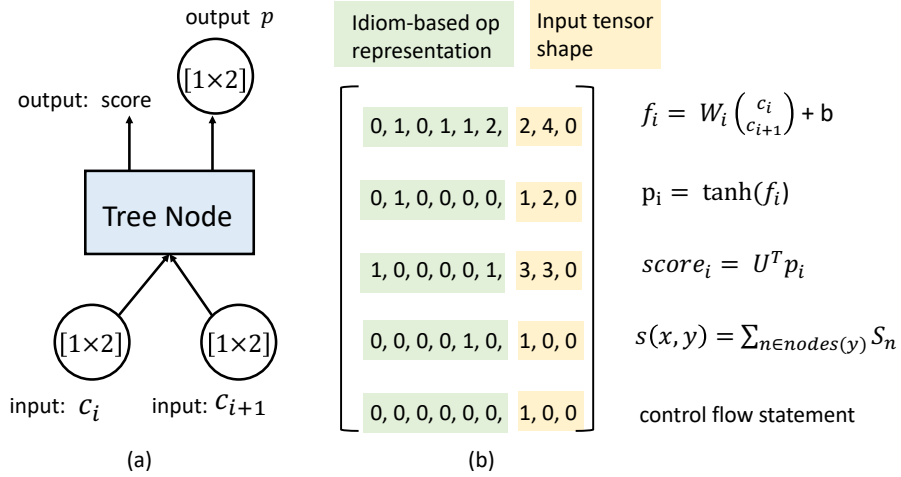


Figure 6.3: An AFM example to show the static structure of the Tree-FC shown in Figure 6.1. (a) A node in DyNN with input and output tensors; (b) AFM representation along with computation in operators.

when the tensor migration should happen, by grouping output operators into *execution blocks*.

Execution blocks. The beginning of an execution block i is the point where tensor migration for the next execution block $i + 1$ starts. The learned model predicts how the operators should be organized into those execution blocks, such that tensor migration for the execution block $i + 1$ can be overlapped with computation in the prior execution block i . The knowledge of how to partition the training of DyNN into execution blocks is learned through training (Section 6.4.4).

Output format. The output of the learned model is multiple vectors, each of which includes operator information for one execution block. Each vector has ten elements: the total number of operators and control flows in the execution block, the numbers of the six idioms accumulated from all operators in the execution block (including six numbers), and the dimension sizes of input/output tensors accumulated from all operators in the execution block (including three numbers). Given the above output and idiom-based representation for DyNN (see Section 6.4.1), we can deterministically resolve dynamism (i.e., the control flows), discussed as follows.

Map output to operators. From the first operator, DyNN-Offload traverses the static architecture of the DyNN, meanwhile bookkeeping the number of operators, the number of idioms, and the dimension sizes of input/output tensors. Whenever a control flow is encountered, DyNN-Offload enumerates and traverses each possible branch. The above traverse continues till the last operator in the DyNN. Whenever any traverse path leads to a bookkeeping record matching the output of the learned model, then that traverse path (including the resolved control flows) is picked to decide tensor migration.

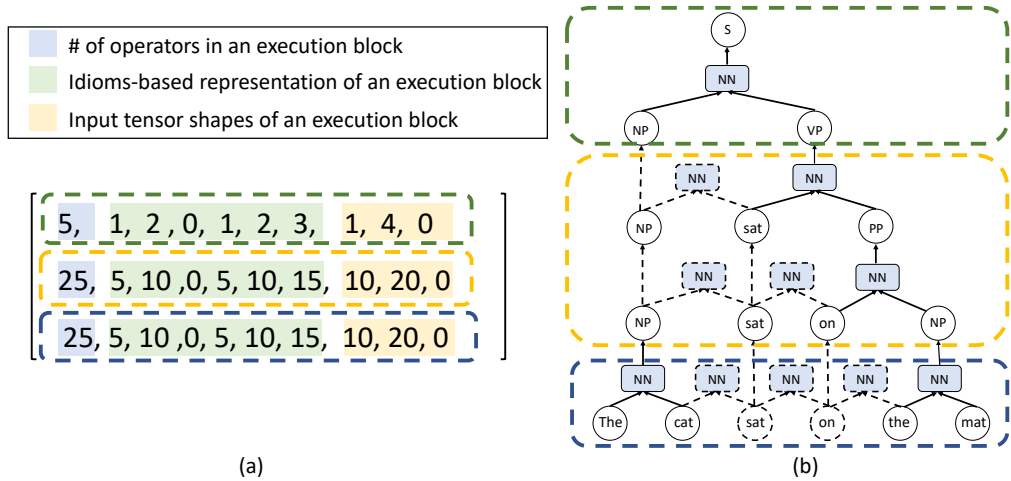


Figure 6.4: An example of the learned model output. (a) shows an output of the learned model with three execution blocks. (b) shows the dataflow graph of DyNN with a given input, and how the output maps back to the operators.

There should always be a traverse path whose bookkeeping record matches the output of the learned model. During the training of the learned model, all the training samples are created to have such a match. Hence, the learned model is expected to generate an output matching a traverse path. In our tests of 15,000 cases, all find an exact match. If the traverse process cannot find an exact match, then DyNN-Offload picks up a traverse path whose bookkeeping record is the closet to the output of the learned model. In particular, among all paths, DyNN-Offload picks up a path where the number of operators is the closet to that in the output of the learned model.

The above process of mapping the output to operators happens at runtime before the DyNN consumes the training sample, but it does not cause large runtime overhead. This is because a large NN model does not have many control flows. In our evaluation, we meet at most 5 traverse paths in a DyNN model. We evaluate the overhead in Section 6.6.3.

An example of the output of the learned model. Figure 6.4 shows an output example for the DyNN example depicted in Figure 6.1. Figure 6.4.(b) shows the dataflow graph corresponding to the DyNN where dotted lines and solid lines show input-dependent dynamic architecture and static architecture respectively. There are three execution blocks in this example.

6.4.3 Light Neural Network Model

Model topology. The learned model includes an embedding and a fully connected NN with only four layers, including one input layer, two hidden layers, and one output layer. All the neurons are regular linear neurons ($y = wx + b$). The inference time of the learned model is

about 10us-15us, which is lightweight.

The embedding converts each DyNN's input sample into a fixed-length vector (128 Dim) as the input of the following fully connected NN. The embedding works with input samples with various sizes/lengths, and serves as a form of feature extraction. The embedding simplifies the input layer of the fully connected NN. For different types of input samples, we use different embedding. In particular, we use Glove [161] for word embedding and Autoencoder for image embedding.

The input layer is supplied with 128 features including AFM and DyNN input sample after embedding. The architecture of DyNN is converted to the feature format (AFM) in an offline way for training. The two hidden layers consist of 512 regular neurons. The two hidden layer use RELU activation functions for its low computation cost and ability to support non-linear modeling. The output layer has 160 neurons with linear activation functions to represent the partition of execution blocks. Overall, there are 800 neurons.

Different models for different types of DyNNs. There are a variety of DyNN architectures, built upon convolutional neural networks (CNN), LSTM, or transformer, etc. Building a general learned model to handle all DyNN architectures is challenging: our evaluation show that a general learned model with an inference time target of $5\mu s$ achieves accuracy of less than 60%, which causes frequent mis-prediction and more than 25% throughout loss because of on-demand tensor migration. We could improve accuracy by adding more layers into the learned model, but that leads to longer inference time. Furthermore, using a general learned model raises challenges on model training and may lack flexibility to be adaptive to new DyNN architectures.

To address the above problem, we classify DyNN in terms of its model architecture into three types, and build a learned model for each type. In particular, the three types are CNN-based DyNN, LSTM-based DyNN, and transformer-based DyNN. The user chooses which learned model should be used based on the user's DyNN architecture. The above method simplifies the process of training the learned model: the user does not need to build a complicated learned model, and it is easier to reach high prediction accuracy (compared with training a general learned model). Although the above method may cause longer training time to train multiple learned models, the training happens offline and its cost can be amortized when the learned model is repeatedly used for various DyNNs with the same type.

6.4.4 Model Training

A training sample is a pair of a input vector and an output vector. The input vector includes AFM for a DyNN and an input to the DyNN. The output vector (or label) uses the same format as the output of the learned model, including operator information for execution blocks (see Section 6.4.2). We discuss how to collect training samples in this section.

Training sample collection. AFM for a DyNN is independent of DyNN’s input and built using static analysis. In particular, the static architecture of the DyNN is collected offline manually to record operator names, input tensor shapes of operators (represented with variables and resolved at runtime), and control flows to build AFM. But this procedure can be replaced by a static analysis tool for Python [166]. In training samples to train the learned model, different DyNNs use different AFMs.

DyNN-Offload collects execution information of DyNN to generate labels for training samples. In particular, given a DyNN and an input to the DyNN to produce a training sample to train the learned model, DyNN-Offload runs the DyNN and generates a dynamic execution trace. The trace includes execution order of operators, their names, input tensor shapes of each operator, and execution time of each operator. This execution trace is used to generate a label.

Labeling. The execution trace of a DyNN gives enough profiling information for a tensor-offloading method for static NN to decide the partition of a dataflow graph. We use the tensor-offloading method in Sentinel [178] because of its generality and short turnaround time. Using a GPU memory capacity, tensor profiling information (including execution order and execution time of operators), and NN topology as input, Sentinel partitions the dataflow graph to maximize the overlap between tensor migration and training computation without violating the GPU memory capacity. DyNN-Offload transforms the output of Sentinel into a representation compatible with output format of the learned model. This new output is used as an output vector (or a label) of a training sample.

To train the learned model, the GPU memory capacity to be used in production environment must be known to generate training samples. This limits the generality of the learned model, but since there are only a handful of available GPU memory capacities in production, a learned model can still be applicable to a number of different GPUs.

6.4.5 Runtime Design

Model inference. The learned model runs on CPU. When a batch of training samples is about to be transferred to GPU to train the DyNN, the learned model is applied to each training

sample in parallel. The output vectors of the learned model for each training sample is sent to the runtime system of DyNN-Offload on CPU to guide tensor migration for all training samples in the batch. The learned model takes the static architecture of DyNN as input. The static architecture is collected offline (discussed in Section 6.4.4).

Memory management and tensor migration. GPU memory is partitioned into two equal-sized buffers: one, called the working buffer, is used for working tensors referenced by the ongoing execution block, and the other, called the migration buffer, is used for prefetching tensors referenced by the next execution block. The two buffers switch roles once the ongoing execution block is done. If tensor prefetching for the next execution block is not complete when the ongoing execution block is done, DyNN-Offload must wait for the completion of prefetching to avoid data race. This double buffering-based mechanism aims to make the best efforts to hide tensor migration overhead.

Since CPU triggers tensor migration for execution blocks and GPU performs execution block-based training, there must be a synchronization mechanism between CPU and GPU. We introduce an operator counter at CPU to record the number of operators (GPU kernels) launched on GPU. When the counter reaches the number of operators in the ongoing execution block i , CPU is aware that GPU starts to execute the execution block $i + 1$, and starts to migrate tensors for $i + 2$.

The migration buffer must evict unused tensors and prefetch tensors used by the next execution block. Eviction and prefetching could happen in parallel to make better utilization of interconnect bandwidth between CPU and GPU. However, we find this solution has difficulty to migrate tensors into a contiguous memory space in GPU, leading to memory fragmentation. Hence, DyNN-Offload evicts tensors first, and then prefetches tensors.

The above memory management methods are implemented at the runtime system of a training framework (e.g., Pytorch). Hence, there is no need to change the DyNN model.

Handling mis-prediction. The learned model may mis-predict operator execution. As a result, when an operator is about to be executed on GPU, an tensor needed by the operator may not be on GPU memory. In this case, DyNN-Offload instruments the runtime error due to tensor missing on GPU and migrates the tensors on demand. Furthermore, DyNN-Offload records the mis-prediction case by recording the static architecture and input of DyNN and the correct execution block. This case is used as a training sample to train the learned model and improve its accuracy in the future.

Furthermore, the mis-prediction case is directly used to avoid repeated mis-prediction for other input samples. Some input samples may lead to the same dataflow graph as the input sample

of the mis-prediction case. To identify such an input, given an input sample (called the new input), the output of the learned model (called the new output) is compared with the output where there is the mis-prediction. If the two outputs are the exactly same, then during the process of mapping the new output to operators, the correct execution block in the mis-prediction case is used to resolve the control flows.

Using the output similarity of the learned model to determine the similarity of dataflow graphs may lead to a false positive decision, i.e., although the two outputs are the same, the corresponding inputs cause different control flows in the data flow graph. However, having such a false position decision is rare, because the chance that the two outputs are the exactly same is very low given high dimensionality of the output. If a false positive decision does happen, using the mis-prediction case to resolve control flows can cause a mis-prediction again, which is recorded. If the false positive decision happens frequently, then the control flow decision in the false positive case is used to avoid mis-prediction for other input samples.

To avoid large runtime overhead of detecting the similarity of dataflow graphs, given an input sample, DyNN-Offload only selects the top three most frequent occurrences of mis-predication cases for comparison.

Impact of dynamic batching. Batching of training samples is commonly used to improve GPU utilization. For DyNN, a batch is dynamically formed by batching operators from multiple dataflow graphs (each dataflow graph corresponds to one training sample). Dynamic batching couples the execution of multiple dataflow graphs, but does not impact the effectiveness of DyNN-Offload, because of two reasons.

(1) Dynamic batching does not change the execution order of execution blocks in each dataflow graph. Hence, the tensor prefetching order predicted by the learned model is still correct. The operator counter-based approach (Section 6.4.5) can still effectively set up synchronization between CPU and GPU for tensor migration. (2) Dynamic batching can extend the execution time of batched operators because of extra cache misses caused by thread block scheduling [225] and TLB misses [45]. But the extended execution time of execution blocks gives more opportunities to overlap with tensor migration. Hence the effectiveness of DyNN-Offload is not compromised.

6.5 Implementation

DyNN-Offload includes (1) a runtime system and (2) an offline training system to collect training samples to train the learned model. The runtime system is implemented on top of ONNX Runtime [147]. Since ONNX Runtime supports a variety of machine learning frameworks (e.g.,

```

0 import torch
1 import torch.nn as nn
2 from ort_support.offload as ort_support
3
4 device = ort_support.set_offload_device()
5 model = BuildModel(config)
6 model = ort_support.create_ort_trainer(device,model)

```

Figure 6.5: An example of using DyNN-Offload.

PyTorch and TensorFlow), operating systems (e.g., Linux and Android), and hardware platforms, DyNN-Offload can benefit various DyNNs regardless of their deployment environment. DyNN-Offload uses a static analysis tool [35] based on LLVM to get and counter idioms in operators. The runtime system has two components: tensor manager and runtime scheduler.

Tensor manager is in charge of tensor (de)allocation and handling of mis-prediction. DyNN-Offload intercepts tensor allocation API `AllocatorDefaultAlloc()` used for GPU memory allocation, and redirects it to CPU memory. DyNN-Offload initially allocates all tensors on CPU memory to avoid out-of-memory errors. To avoid memory leak, DyNN-Offload intercepts tensor `AllocatorDefaultFree()` to ensure memory space is freed regardless of the location of the tensor. Furthermore, DyNN-Offload implements a tensor fault handler leveraging the tensor hook mechanism in ONNX. The tensor fault handler is invoked when any tensor needed by GPU computation is missing in GPU memory and a `cudaErrorInvalidAddressSpace` fault is reported. The handler fetches the missing tensor from CPU memory and records the mis-prediction information to a file.

Runtime scheduler is used for the learned model inference and runtime tensor migration. In particular, the learned model is implemented as a user-defined operator `learned_model_inference()`, which is used as the first operator in a dataflow graph for DyNN. `learned_model_inference()` takes an input sample of DyNN and runs the learned model on CPU. The generated AFM is used in ONNX runtime scheduling. Based upon the ONNX runtime to launch operators, DyNN-Offload counts the number of launched operators and triggers tensor migration asynchronously. Within an execution block, DyNN-Offload migrates tensors without priority. DyNN-Offload waits for the completion of tensor migration and starts the computation for the next execution block.

Figure 6.5 illustrates how to use DyNN-Offload. DyNN-Offload is transparent to data scientists and does not require model refactoring. Only Line 4 (deciding hardware target for offloading) and Line 6 (enabling training) need to be added.

Training system for the learned model includes (1) an execution trace generator, (2) a partition simulator, and (3) a training sample generator.

The *execution trace generator* is based upon existing tensor instrumentation infrastructures

Table 6.2: DyNNs for evaluation.

Model Name	Model Type	Dataset	Batch Size
Tree-CNN [185]	CNN	CUB 200	64
UGAN [134]	RNN	circular gaussian	1024
Tree-LSTM [200]	LSTM	SICK	512
var-LSTM [124]	LSTM	Reuters-21578	512
fixed-LSTM [199]	LSTM	Reuters-21578	512
var-Bert [150]	transformer	wikitext-2-v1	16
fixed-Bert [43]	transformer	wikitext-2-v1	16

in Pytorch or TensorFlow to generate the dynamic execution trace in a Json-formatted file. The *partition simulator* consumes this file and implements the partition algorithm in Sentinel. The partition algorithm specifies where tensor migration should be triggered in terms of DyNN model topology. The partition simulator transforms this partition result into a representation compatible with the output of the learned model by aggregating profiling results between partitions. The *training sample generator* pairs up the outputs of the partition simulator (i.e., labels), AFMs for DyNNs, and DyNN input samples to generate training samples to train the learned model. We train three learned models, targeting on CNN-, LSTM-, and transformed-based DyNNs.

6.6 Evaluation

6.6.1 Methodology

Experimental setup. Our experiments were conducted on a server equipped with NVIDIA RTX6000 GPUs (GPU for desktop) with 23GB memory, and dual Intel Xeon CPUs 2.6GHz (totally 24 cores) and 186GB CPU memory. The interconnect between CPU and GPU is 16-line PCIe 3.0. We use Ubuntu 18.04, CUDA Toolkit 11.2, and PyTorch 1.10.0.

Workloads. We evaluate DyNN-Offload with five DyNNs and two static NNs (fixed-LSTM and fixed-Bert). See Table 6.2. Their memory consumption is larger than the GPU memory capacity. To train the learned models, we use the training datasets coming with the CNN-, LSTM-, and transformer-based DyNNs to generate training samples. We build 5,000 samples to train each learned model, and 1,000 samples to test/evaluate the performance of learned model. Training and testing samples do not have any overlap.

Baselines for evaluation are summarized as follows.

- *DTR* [100] is a state-art-the-art solution based on tensor rematerialization. DTR frees memory space for activation tensors when the GPU memory is not large enough. DTR rematerializes the freed activation when needed.

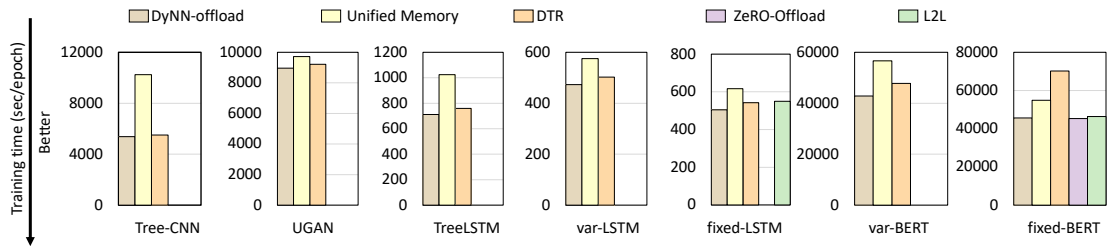


Figure 6.6: Performance comparison between existing solutions and DyNN-Offload with seven workloads.

- *Unified memory* [146] enables memory oversubscription by migrating pages on demand and using limited user hints.
- *L2L* [164] is an industry-quality solution from Microsoft, aiming for saving GPU memory for static NN based on PGO. L2L migrates parameters, activations, and optimizers layer by layer between CPU and GPU memories.
- *ZeRO-Offload* [179] is an industry-quality solution from Microsoft, aiming to optimize tensor offloading for static transformer models to save GPU memory based on PGO.

6.6.2 Overall Performance

Figure 6.6 shows the training time with DyNN-Offload and the baselines. We report the training time for one epoch after warmed-up run. ZeRO-Offload can only work for transformer models (i.e., var-BERT and fixed-BERT). L2L can only work for static models. In general, DyNN-Offload consistently outperforms other solutions: by 2% - 50% for dynamic models, and by 11% - 34% for static models. We further see that:

- Unified memory performs worst in almost all cases, because most of tensor migration happens on demand.
- DTR performs much worse than DyNN-Offload by 75% for fixed-BERT, because large activations in BERT leads to large rematerialization cost. DTR performs similar to DyNN-Offload for Tree-CNN and UGAN, because re-materialization cost for the two models is small.
- DyNN-Offload performs similarly to ZeRO-Offload and L2L (see fixed-BERT and fixed-LSTM). ZeRO-Offload and L2L are highly-optimized based on extensive workload characterization and machine learning domain knowledge. Without those expensive optimizations, DyNN-Offload is able to achieve the similar performance, but is more general.

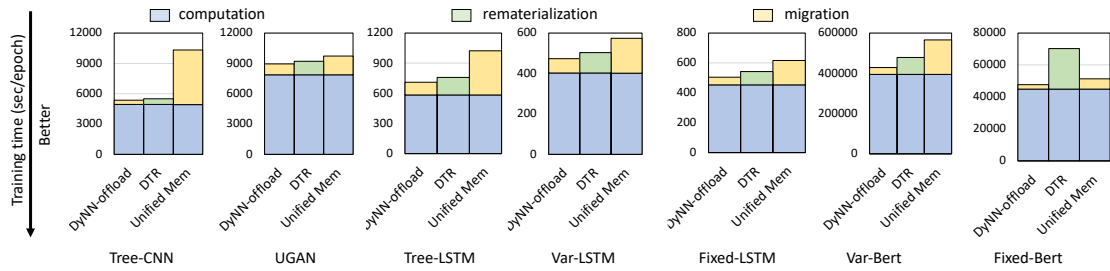


Figure 6.7: Performance breakdown for DyNN-Offload, DTR and unified memory.

6.6.3 Performance Analysis

We break down execution time to further analyze performance. Figure 6.7 show GPU computation time, rematerialization time in DTR, and tensor migration time exposed to the critical path. We have three observations. (1) Unified memory spends a large portion of time on tensor migration, which is especially pronounced in Tree-CNN and Tree-LSTM where tensor migration takes 55% and 40% of training time. (2) Rematerialization is costly. In fixed-Bert, it takes 33% of training time. rematerialization in var-Bert is cheaper than in fixed-Bert, because the depth of recursive rematerialization in var-Bert is shallower. (3) By removing rematerialization and hiding migration, DyNN-Offload outperforms both the solutions.

Overhead analysis. The overhead of DyNN-Offload includes (1) the inference time and (2) the time of mapping the output of the learned model to operators. For (1), the inference time is 12 μs . For (2), the time is about 10-15 μs . The overhead is much smaller than the iteration time of NN models we evaluate (i.e., a few milliseconds to a few seconds).

6.6.4 Scalability of DyNN-Offload

We evaluate scalability from two perspectives: (1) performance when increasing the complexity of DyNN (Figure 6.8); (2) performance with larger system scales (Figure 6.9).

For (1), we use var-Bert (a dynamic transformer model), and vary the number of transformer blocks in its static architecture. To evaluate DyNN-Offload, we compare with DTR. Figure 6.8 shows as the model scale increases, the performance benefit of DyNN-Offload over DTR becomes larger. As the DyNN becomes larger, the DTR overhead (the rematerialization cost) becomes larger, offsetting its performance benefit.

For (2), we use data parallelism on six machines, each of which has an NVIDIA RTX6000 GPU. Figure 6.9 shows the results. As the scale becomes larger, the training throughput increases

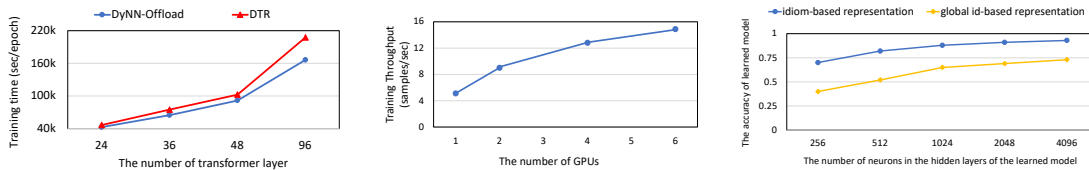


Figure 6.8: Scalability evaluation of DyNN-Offload with a variety of model scales in var-Bert. Figure 6.9: Scalability evaluation of DyNN-Offload in terms of system scales (i.e., the number of GPUs). Figure 6.10: Evaluation of the effectiveness of the idiom-based representation.

Table 6.3: Learned model performance with different model complexity. “LM” stands for learned model.

LM Complexity (# of neurons)	LM Accuracy	LM Training Time	LM Inference Time	LM Memory Consumption
256	0.7	2h	5us	40KB
512	0.82	2.5h	12us	75KB
1024	0.88	3h	20us	140KB
2048	0.91	4h	42us	260KB
4096	0.93	6h	80us	530KB

proportionally until four GPUs. After that, the performance scaling slows down because of increasing inter-GPU communication cost, but the overhead of DyNN-Offload and on-demand tensor migration caused by mis-prediction remain constant at all scales, showing good scalability.

With DyNN-Offload, PyTorch is able to train var-Bert with 96 transformer layers (1.4B parameters, which is the largest model in our evaluation), on our GPU (23GB memory) without any out-of-memory error. But without DyNN-Offload, PyTorch is only able to train var-Bert with 12 transformer layers (170M parameters), a 8x reduction in terms of model size.

6.6.5 Construction of Learned Model

We study how to construct an effective and efficient learned model. *We want to reduce inference time as much as possible* but with high modeling accuracy. This is especially important in use scenarios (e.g., mobile devices) where DyNN has short iteration time. We change the number of neurons in the hidden layers and study its effect, shown in Table 6.3. When increasing neurons from 256 to 512, the model accuracy largely increases by 0.12. However, when we increase neurons beyond 512, the momentum of accuracy increase is significantly smaller, but the inference time continuously increases with a rate of approximately 2x. Hence we choose 512 for our learned model, because of its good balance between accuracy and costs (including inference and training times).

The number of mis-prediction cases. With 512 neurons in the hidden layers, we find less

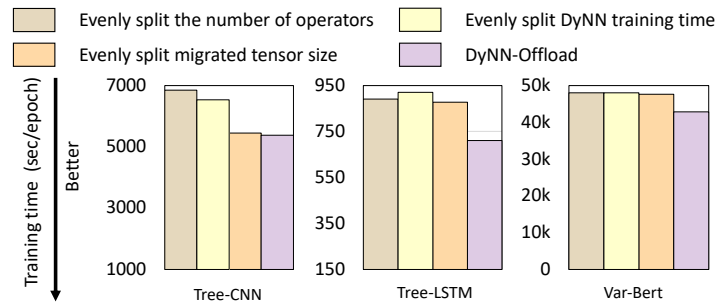


Figure 6.11: The training time with DyNN-Offload and three heuristic solutions to partition DyNNs.

than 60 mis-prediction cases in each of the seven NN models with 3,000 testing samples.

6.6.6 Idiom-based Representation

We compare idiom-based representation with another solution based on operator type. We give each operator type a unique ID and name this approach “global id-based representation”. Using the two representations, we train two learned models. Figure 6.10 shows the accuracy. Given the same model complexity (in terms of the number of neurons), the idiom-based representation outperforms the global id-based one in terms of accuracy by at least 19%. To reach the same accuracy, for example 0.73, the idiom-based one needs 256 neurons, while the global id-based one needs 4096, which increases the inference time by x6 and training time by 2.4x. This is because using the operator type largely increases the input feature space, and has to increase model complexity to improve accuracy.

6.6.7 Evaluation of DyNN Model Partition

DyNN-Offload partitions the dataflow graph of DyNN into multiple execution blocks to hide tensor migration. We evaluate the partition method in DyNN-Offload with three heuristics: (1) one partitions the DyNN by evenly splitting the number of operators, (2) one by evenly splitting the training time, and (3) one by evenly splitting the size of all tensors. These solutions use the same number of partitions as DyNN-Offload. Figure 6.11 shows the results for three DyNN models. DyNN-Offload outperforms the three solutions by 2% - 29%, because DyNN-Offload can adaptively change the partition size to hide tensor migration based on the learned knowledge while the other solutions use the fix-sized partitions.

6.6.8 Impact of Handling Misprediction

We study the number of mis-prediction cases without and with handling them. DyNN-Offload handles mis-prediction cases by using them to avoid repeated mis-prediction for some cases and improve the model accuracy of the learned model. Without handling mis-prediction, the number of mis-prediction for Tree-CNN, Tree-LSTM, and var-Bert is 167, 109, 182 respectively, evaluated with 3,000 training samples (cases). With handling mis-prediction, the mis-prediction number for Tree-CNN, Tree-LSTM, and var-Bert decreases to 59, 42, 102.

6.7 Related Work

Machine learning for memory and storage. LinnOS [71] uses a light NN for inferring SSD performance at per-IO granularity to achieve performance predictability. KML [15] and LearnedSSD [110] employ machine learning to tune storage configurations. LLAMA [126] introduces NN to predict data object lifetime and avoid memory fragmentation. Cori [49] and Kleio [48] use machine learning to decide page migration frequency and granularity on heterogeneous memory.

Several recent efforts use neural network for prefetching. Voyager [194] builds hierarchical neural networks to learn address correlation and prefetch irregular sequences of memory accesses. Peled et al. [153] use a table-based reinforcement learning (RL) framework to explore the correlation between program contexts and memory addresses. Hashemi et al. [73] formulate prefetching as a classification problem and use LSTM as a prefetcher. DyNN-Offload is different from the existing work, because it focuses on a unique memory capacity problem for training DyNN and customizes the learned model based on the characterization of DyNN.

System supports for DyNN focus on batching dynamic dataflow graphs to improve hardware utilization. Since different input samples use different dataflow graphs, batching them together with unresolved control flows is challenging.

TensorFlow Fold uses a depth-based batching [124], which dynamically batches nodes with the same depth and shapes (in terms of tensor dimensions) in multiple dataflow graphs. However, this method misses good batching opportunities (e.g., the loss functions in different dataflow graphs can be at different depths and cannot be batched). DyNet [139] uses an agenda-based batching that dynamically tracks nodes with dependencies resolved for batching. However, this method focuses on individual nodes and is not open to dataflow graph level optimizations. Cavs [233] represents DyNN with a static vertex function and a dynamic instance-specific graph.

The scheduling of the static function exposes batched execution opportunities over multiple input samples. However, Cavs needs many programming efforts. DyNN-Offload could complement the above efforts by providing a new method for batching. By transforming dynamic graphs into static ones based on the learned model, DyNN-Offload enables batching and graph level optimizations without changing DyNN.

Input-aware performance optimization has been utilized for input-sensitive applications, such as streaming graph processing [27], Spark programs [242], sorting [47], and sparse matrix multiplication [232]. A common theme of the above work is to use input knowledge to determine how to optimize performance (e.g., deciding configurations for autotuning or computation granularity for aggregation). Besides that, input knowledge has been used for high-performance code generation for GPU [127, 188, 205]. Different from the existing efforts, DyNN-Offload recognizes the implicit knowledge in input samples about how the dataflow graph will be executed and hence can be utilized to save GPU memory.

6.8 Conclusion

DyNN-Offload is a memory management system enabling large DyNN training with limited GPU memory capacity. Unlike the traditional PGO-based approach that lacks abilities to react to dynamism in DyNN, DyNN-Offload uses a learned approach to resolve dynamism and predict access order of tensors. We have show the feasibility of building a fast, accurate, and live machine learning model to guide performance optimization and analysis for machine learning systems.

Chapter 7

Optimizing Large-Scale Plasma Simulations on Big Memory System

Particle simulations of plasma are important for understanding plasma dynamics in space weather and fusion devices. However, production simulations that use billions and even trillions of computational particles require high memory capacity. In this work, we explore the latest persistent memory (PM) hardware to enable large-scale plasma simulations at unprecedented scales on a single machine. We use WarpX, an advanced plasma simulation code which is mission-critical and targets future exascale systems. We analyze the performance of WarpX on PM-based systems and propose a hybrid of static and dynamic data placement for performance optimization. We develop a performance model to enable efficient data migration between PM and DRAM in the background, without reducing available bandwidth and parallelism to the application threads. Our design achieves 64.6% performance improvement over the PM-only baseline and outperforms DRAM-cached, NUMA first-touch, and a state-of-the-art software solution by 34.4%, 41%, and 83.3%, respectively.

7.1 Introduction

Plasma simulations are critical for understanding plasma dynamics in space weather and fusion devices [30, 210, 220]. The particle-in-cell (PIC) method is an important model that enables large-scale plasma simulations on high-performance computing (HPC) systems [32, 66, 208, 220]. The PIC method uses computational particles to simulate plasma particles, such as electrons and protons. High-fidelity PIC simulations often use billions and even trillions of particles, which

require high memory capacity.

Persistent memory (PM), exemplified by the Intel Optane DC PM [87], provides a solution to meet the requirement of high memory capacity in HPC applications. For instance, the Intel Optane PM can provide up to six terabyte (TB) memory on a single machine. However, there is a performance gap between PM and DRAM [87, 155]. Read and write bandwidth of the Optane PM are only 38% and 16% that of DRAM, respectively. Hence, PM often comes with a small DRAM (tens of gigabytes) to boost performance. As a result, PM and DRAM form a heterogeneous memory (HM) system. How to place and migrate data between PM and DRAM to enjoy the speed of DRAM and capacity of PM remains active research [56, 77, 140, 154, 180, 226, 227].

In this paper, we leverage the latest PM hardware to enable large-scale plasma simulations. We analyze the performance and develop a performance model for optimizing PIC codes on PM-DRAM systems. Our performance analysis and optimization use a state-of-the-art electromagnetic PIC code called WarpX [208]. Nonetheless, the optimization strategies derived from this work are generally applicable to other PIC-based simulation codes.

WarpX [208] is a mission-critical application designed for efficient executions on large-scale HPC systems and future Exascale machines. WarpX enables high-fidelity modeling of many complex processes, such as laser- and beam-driven plasma accelerators. As a PIC method, WarpX has high memory footprints for simulating particles moving in electromagnetic fields. The memory footprint scales up with the number of particles and field size. For example, the recent production run on 4,096 nodes on the Cori supercomputer simulates 62 billions of particles and consumes up to 8.9 TB memory. Therefore, a large memory capacity is a key enabler for large-scale simulations in WarpX.

Our performance analysis identifies two challenges in optimizing WarpX on PM-based systems. First, WarpX has frequent read/write with a streaming-like access pattern, which intensifies memory accesses. Given the low bandwidth of PM compared to DRAM, this access pattern is unfavorable. Second, the WarpX code uses tens of millions of data objects and frequent memory (de)allocation. These data objects include long-lived data structures for particles, fields, and metadata, as well as short-lived buffers for communication and computation. Managing such a large number of data objects with diverse properties on DRAM and PM is complex.

We introduce a set of techniques to optimize the performance of WarpX on PM. Data objects are characterized and classified based on their lifetime and memory access patterns. This information guides their placement and migration on PM and DRAM at runtime. Ideally, frequently accessed data objects are placed into DRAM. However, due to the limited DRAM

capacity and the large problem size in production runs, only some data objects or even partial data objects can fit into DRAM. To address this challenge, we partition long-lived large data objects and migrate their partitions between PM and DRAM. To achieve efficient migration, we need to address two challenges. First, migrating data between PM and DRAM consumes memory bandwidth. However, the application also needs to access memory. Hence, data migration can compete with the application threads for memory bandwidth. Second, data migration uses helper threads in the background, other than the application threads, to avoid exposing data migration into the critical path. However, using helper threads reduces the availability of processor cores for the application threads. An optimal number of helper threads should expedite data migration without causing performance loss in the application threads.

To address the above challenges, we develop a performance model to decide the optimal number of helper threads for data migration. Our model considers the constraints on memory bandwidth and core availability in realistic simulations. Based on the performance model, we use a lightweight runtime algorithm combined with runtime profiling and empirical observations to select and adapt the data migration between PM and DRAM for different input problems.

We summarize the paper contributions as follows.

- We demonstrate and quantify the benefits of leveraging PM to enable large-scale plasma simulations in a mission-critical application called WarpX.
- We characterize the memory management, bandwidth consumption, and data object lifetime and access patterns in WarpX production simulations. We analyze the implication of the characterization for performance optimization on PM-based systems.
- We propose static and dynamic data placement strategies and develop a performance model for efficient data migration between PM and DRAM.
- We improved the WarpX execution on Optane-only by 64.6% and outperformed DRAM-cached, the NUMA first-touch policy, and a state-of-the-art HM solution by 34.4%, 41% and 83.3%, respectively.

7.2 Background

The WarpX particle-in-cell code. WarpX leverages MPI+OpenMP parallelism. It has two components, i.e., PICSAR [163] for particle-in-cell (PIC) routines at the innermost level and AMReX [247] for adaptive mesh refinement (AMR). A WarpX simulation may consist of multiple

levels of resolution. Each level is an AMR level in the AMReX library and performs a PIC simulation at the resolution of that level.

PIC codes typically have the following characteristics. Field and particles are the main data structures, and particles consume the most memory footprint. The core PIC routines include four phases—`current deposition`, `field solver`, `field gather`, and `particle pusher`. In `current deposition`, all particles are iterated to deposit their charge and moments to the fields. In `field solver`, a linear system from the discretized Maxwell’s equations is solved to compute electric and magnetic fields on the grid. During `field gather`, forces from the fields are calculated for each particle, which then in `particle pusher`, are used to update the location of particles. Both `current deposition` and `field gather` have mostly regular data access to the particles, exhibiting streaming-like read access in `current deposition` and read-write access in `field gather`.

Communication happens in `field solver` and `particle pusher`. Most communication in `field solver` is point-to-point (P2P) between neighbor processes for halo exchange. Both collective and P2P communications are used in `particle pusher` for communicating particles that move from one subdomain to another.

The Intel Optane DC PM. The Intel Optane DC Persistent Memory Module (PMM) is the first large-scale byte-addressable PM. The Intel Purley platform used in our study is equipped with Optane PM DIMMs and DRAM DIMMs. Each socket has six memory channels, and each is shared by a DRAM DIMM and a PMM DIMM. In total, there are 12 PM DIMMs and DRAM DIMMs, respectively, on two sockets. An Optane PM DIMM may have 128, 256, or 512 GB capacity, enabling up to 6 TB memory capacity on a single machine [87]. The latency to PM is measured as 174 ns for sequential reads and 304 ns for random reads, in contrast to 79 ns and 87 ns to DRAM [155]. The bandwidth to PM on one socket is 39 GB/s for read and 13 GB/s for write, while DRAM achieves 104 GB/s and 80 GB/s bandwidth on the same platform. There are two modes in PMM: *memory mode* and *app-direct mode*. In the memory mode, DRAM becomes a hardware-managed cache to PMM. Running the application on DRAM-cached PMM to use both DRAM and PMM requires no application modifications. In the app-direct mode, accesses to PM and DRAM can be explicitly controlled at the application level, either through a DAX-based file system [187] or exposing PM as separate NUMA nodes.

Enabling Large-Scale Simulations with PM. Using the Optane persistent memory, we can significantly increase the memory capacity per node to enable fine-grained and large-scale scientific simulations. An Optane-based machine has up to six TB memory [87], while a node in

Table 7.1: Compare the memory capacity and simulation scale on supercomputers

Supercomputer	Mem capacity per node	Largest problem (in terms of particles)
Sierra	320GB DRAM	10.6 trillions
Summit	608GB DRAM	18.9 trillions
Aurora	256GB DRAM (est.)	8.8 trillions
Taihu Light	32GB DRAM	1.1 trillions
Optane-based	1692GB (1.5TB PM + 192GB DRAM)	58.6 trillions

main-stream supercomputers has at most hundreds of GB (see Table 7.1). Given a fixed number of nodes, using the Optane PM allows us to perform scientific simulation previously unachievable due to limited memory capacity.

Table 7.1 presents an example case that performs a numerical simulation of a laser-driven plasma accelerator (i.e., the laser-wakefield accelerator) using WarpX [57]. This simulation uses a larger number of particles in the time and space scales to gain knowledge on plasma structures towards a full-scale numerical study of the next generation laser-wakefield accelerator systems. Such a numerical study provides insights for more-compact high-energy colliders [105].

In this example, we assume the same simulation configuration as that in a production run on 4,942 nodes on the Cori supercomputer. Table 7.1 compares the largest simulation scale that can be supported on each supercomputer. The simulation scale is defined as the number of simulated particles – a larger number indicates a larger simulation scale. Clearly, *Memory capacity* is one main constraint on the simulation scale. The memory consumption of WarpX is calculated based on the estimation of the sizes of particles, fields, metadata, and temporal data objects.

Table 7.1 shows that an Optane-based supercomputer can enable larger-scale simulations than other supercomputers. Compared with Summit and Sierra (the top two supercomputers in the top500 list by April 2020) that use hundreds of Gigabytes of DRAM per node, the Optane-based supercomputer increases the simulation scale by 3.1x and 5.5x, respectively.

7.3 Performance Characterization

We develop a heap profiler and a phase profiler to characterize the memory usage and bandwidth consumption in the application. The heap profiler tracks dynamic memory allocations and collects information on each allocation (data object). The phase profiler collects hardware events from performance counters and associates them to specific execution phases in the application.

Heap profiler interposes common memory management routines in C and C++, e.g., `malloc`, `calloc`, the operator `new` and its variants, `posix_memalign`, Linux-specific `aligned_alloc`

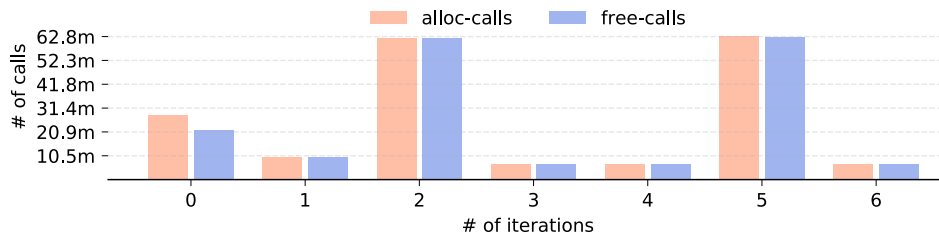


Figure 7.1: The number of memory allocation/deallocation across iterations.

and `valloc`. It collects the metadata of data objects, including size, time of allocation/deallocation, and lifetime (defined as the interval between allocation and deallocation). The timestamps of allocation and deallocation are used to map to specific execution phases of the application. The tool also supports postmortem analysis of the profiling results.

Phase profiler use specific APIs to track execution phases. The user inserts the APIs into the WarpX code to mark execution phases. The API implementation includes two functionalities. First, it triggers a set of auxiliary external scripts to invoke the Linux performance profiling tool `perf` to collect information from hardware performance counters. Also, it invokes the Intel PCM [204] to collect memory bandwidth data.

7.3.1 Profiling Results

We use a representative laser-driven simulation configuration for profiling. The input problem uses $704 \times 704 \times 5664$ cells and 8.4 billion particles (see Problem B in Table 7.5). The peak memory consumption exceeds 1.2 TB on DRAM-cached Optane (memory mode).

Memory allocation and deallocation analysis. We use the heap profiler to track memory allocation/deallocation in each iteration of the WarpX execution. Figure 7.1 presents the results for the first seven iterations. The profiling results show that millions of memory allocation and deallocation occur in each iteration. Across iterations, the number of memory allocation and deallocation varies. Such a massive amount of data objects, which are as resulted from frequent allocation and deallocation, imposes challenges in profiling at either data object level [78, 156, 226] or memory page level [14, 53, 95, 227, 234].

Data object lifetime and size. We classify the distribution of lifetime and size of data objects. Table 7.2 reports the classification in the second iteration of the WarpX simulation. Other iterations exhibit similar distributions. A data object is alive after its allocation and before its deallocation. We categorize a data object as short-lived if its lifetime is within one iteration and long-lived otherwise. We observe that 92.7% of data objects are short-lived in the WarpX simulation. Furthermore, these short-lived data objects only account for less than 10% of the peak

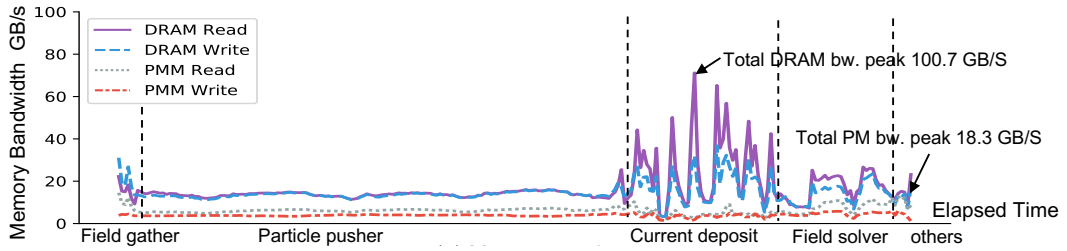


Figure 7.2: Memory bandwidth consumption in major phases.

Table 7.2: The distribution of object size.

Bin (MiB)	Short-lived data object		Long-lived data object	
	Accumulated footprint	Peak footprint	Accumulated footprint	Peak footprint
(0,1)	897.7 GiB	10.4 GiB	840.3 GiB	840.3 GiB
[1,2)	34.3 GiB	10.8 GiB	1.9 GiB	1.9 GiB
[2,4)	262.5 GiB	66.0 GiB	285.5 GiB	285.5 GiB
[4,8)	144.0 MiB	16.0 MiB	543.0 MiB	543.0 MiB
[8,16)	96.0 MiB	16.0 MiB	14.0 MiB	14.0 MiB
[16,32)	192.0 MiB	32.0 MiB	28.0 MiB	28.0 MiB
[32,64)	384.0 MiB	64.0 MiB	0	0
[64,+∞)	768.0 MiB	1.6 GiB	0	0

Table 7.3: The breakdown of execution time.

	Particle pusher	Current deposition	Field solver	Field gather	Others
Ave. time	300.8s	132.0s	47.2s	25.2s	9.9s
Percentage	58%	26%	9%	4.9%	2.1%

memory consumption of WarpX. This characterization motivates us to use a small DRAM space to host repeatedly allocated/freed short-lived data objects and avoid data movement between DRAM and PM. This static placement strategy is described in Section 7.4.1.

Execution time breakdown. We measure the time of major execution phases (Section 7.2). Each iteration of the main computation loop performs these major phases. Some “add-ons” execution (such as load redistribution and moving window) may also occur in some iterations, counted as *others*. Table 7.3 reports the breakdown of the execution time.

Overall, the `particle pusher` and `current deposition` phases account for about 84% of the total simulation time. `Particle pusher` reads the fields and updates the position of each particle. `Current deposition` reads each particle and updates the current densities on fields. These phases dominate the execution time and the read/write accesses to the main memory. Therefore, we employ fine-grained dynamic data management to optimize their performance. We describe the dynamic strategy in Section 7.4.2.

Memory bandwidth analysis. We measure the memory bandwidth in major phases and report in Figure 7.2. We observe that the execution of WarpX is not bounded by DRAM/PM

bandwidth in most of the execution time (e.g., `field gather` and `particle pusher`). When the memory bandwidth utilization is low, e.g., about 10%, prefetching data to DRAM would not constraint the bandwidth used by the application. Thus, performance improvement becomes feasible. However, since data prefetching consumes memory bandwidth, using it in bandwidth-intensive phases (e.g., `current deposit`) may cause performance loss in the application. The bandwidth analysis motivates us to develop a performance model to optimize data prefetching at runtime (Section 7.4.2).

7.4 Performance Optimization on PM

We propose a runtime system, called WarpX-PM (Figure 7.3), to manage data placement on DRAM and PM automatically. WarpX-PM partitions DRAM into four spaces to store data objects with different functionality and access patterns in WarpX. The *metadata space* stores metadata updated infrequently but accessed frequently. The *temporary space* is used for short-lived data objects frequently allocated and freed. Those short-lived data objects share and reuse the temporary space without causing data movement between DRAM and PM. The *migration space* acts as a software-managed DRAM cache to prefetch particles from PM before they are accessed in computation. Finally, the *free space* stores the maximum possible field data.

We combine static and dynamic strategies for data placement in the four spaces. Except for the migration space managed for dynamic data placement, the other three spaces are used for static data placement. We use performance modeling to guide the data copy between DRAM and PM without disturbing the WarpX performance. Our designs are described in detail as follows.

7.4.1 Static Data Placement

WarpX-PM uses static placement to addresses the fundamental limitations in the memory mode. This memory mode uses DRAM as a direct-mapped hardware cache. Consequently, some performance-critical data objects are evicted from DRAM due to iterative accesses to large data objects, such as particles and fields. Examples of performance-critical data objects include metadata and temporary data, where metadata is used to compute the simulation domain iteratively, and temporary data is used to adjust the size of data objects during the computation. These performance-critical data objects are frequently referenced but only consume a small portion (less than 10%) of the total memory consumption. In the memory mode, these data objects are frequently moved between DRAM cache and PM, a typical manifestation of cache thrashing.

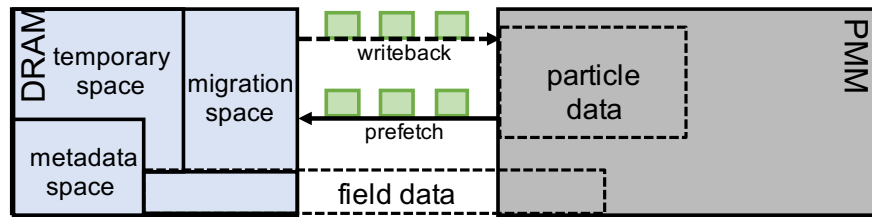


Figure 7.3: The overview of data management on Optane-based HM.

Static data placement takes effect on all execution phases. WarpX-PM pins the performance-critical data objects to DRAM to avoid moving them between DRAM and PM as in the memory mode. Depending on their lifetime, they can be categorized as long-lived and short-lived, and placed into the metadata space and temporary space, respectively. We describe the management of these two kinds of performance-critical data objects as follows.

Long-lived, performance-critical data objects are mostly metadata and are placed in the metadata space in DRAM directly. In WarpX, the whole simulation domain is decomposed into many *boxes* distributed over MPI ranks. Each box contains a fraction of fields and particles. Metadata is used to record the distribution of boxes in the simulation domain. For instance, *FArrayBox* is a part of box metadata for iterating particles in a box. Metadata are allocated before the main computation loop and only freed after the whole computation finishes – long-lived. During their life span, metadata are frequently accessed, and their size remains unchanged.

Short-lived, performance-critical data objects are typically allocated and freed within one iteration of the main computation loop. These data objects include communication buffers and the memory space used for resizing the data objects during the computation. WarpX-PM allocates these data objects on demand in the temporary space, which is a pre-allocated memory space in DRAM. To ensure the pre-allocated temporary space is large enough for all temporary data objects throughout the computation loop, WarpX-PM uses the following algorithm.

WarpX-PM uses the first iteration of a simulation to measure the peak memory consumption of WarpX. Then, WarpX-PM deducts the sizes of particles, fields, metadata, and a fixed buffer per MPI rank for migration space from the peak memory consumption. The resulted size is used to reserve the temporary space. This approach provides an estimation of the peak memory consumption of short-lived, performance-critical data objects. Across iterations, the peak memory consumption of these data objects may vary, mostly due to communication buffers. The variance is typically small (tens of MB). If the temporary space is exhausted, WarpX-PM increases the temporary space on demand to accommodate.

After the metadata, temporary, and migration spaces are allocated, the remaining space in DRAM is used as the free space. WarpX-PM utilizes it to hold as many as possible fields. Fields are frequently accessed in all phases. WarpX-PM chooses fields instead of particles for static data placement because fields are not allocated in contiguous memory space. Hence, maintaining their location information and copying them between DRAM and PM incur large overhead. Besides, fields are smaller but more intensively accessed than particles.

The static data placement completes after the first iteration. The memory allocation overhead is negligible because only three spaces need to be managed. Furthermore, using the pre-allocated temporary space reduces the overhead of frequent memory (de)allocation for short-lived data objects.

7.4.2 Dynamic Data Placement

The dynamic strategy takes effect at `particle pusher`, `current deposition`, and `field solver`. The accesses to particles mainly occur in these phases. The dynamic placement copies particles into DRAM in batches and only copies them back to PM if particles are updated in the computation. Particles consume at least 50% of memory consumption. For a large input problem, particles alone may unlikely fit in DRAM. However, directly accessing particles in PM in particle computation causes performance loss due to the low memory bandwidth. WarpX-PM uses software-managed *particle prefetching* to copy batches of particles into the migration space so that computation always accesses particles in DRAM.

`ParticleContainer` is the primary data object for particles. It contains an array of particle structures, each representing a particle and recording its position, velocities, ID, and the owner CPU. Thus, `ParticleContainer` occupies a contiguous space in physical memory. In each of the `particle pusher`, `current deposition`, and `field solver` phases, all particles in the `ParticleContainer` are iterated in a streaming-like access pattern at the granularity of `FArrayBox`. WarpX-PM leverages this characteristic to partition each phase into intervals based on the time of processing particles in `FArrayBox`. At an interval i , WarpX-PM copies a batch of particles needed for the next interval $i + 1$ to DRAM. This data copy is expected to finish before the interval $i + 1$. If particles are updated in the interval $i + 1$, they are copied back to PM in the interval $i + 2$. Given the streaming-like patterns to access particles, there is no data dependency between intervals.

To implement the particle prefetching strategy, two challenges must be addressed. First, WarpX-PM needs to decide the number of threads to copy particle batches. WarpX-PM uses

Table 7.4: Notation for performance modeling

Source	Symbol	Description
Hardware parameters	$BW^{DRAM_to_PM}()$	BW of copying data from DRAM to PM
	$BW^{PM_to_DRAM}()$	BW of copying data from PM to DRAM
	BW_{max}	Peak memory bandwidth
	$Thrd_{max}$	Maximum number of hardware threads
App related parameters	$data_{in}, data_{out}$	Sizes of data copied in/out of DRAM
	T_{cp}	Data copying time for an interval
	T_{comp}	WarpX execution time of an interval
	$Thrd_{cp}$	Number of threads to copy data
	$Thrd_{comp}$	Number of threads for application
	T'_{comp}	Optimal execution time of an interval

helper threads instead of application threads to copy particles to avoid delaying the execution of application threads. Using a large number of helper threads accelerate data copy but reduces processor cores and memory bandwidth available for WaprX execution. Using a small number of helper threads increases the risk of exposing data copy into the critical path of WaprX execution if data copy cannot finish in time. Second, the decision of the number of helper threads must be adaptive and lightweight. Different input problems or MPI/OpenMP configurations may consume memory bandwidth differently and need different numbers of helper threads for the best performance.

Performance Modeling. We introduce a performance model-based approach to decide the optimal number of helper threads for each phase. All intervals in the same phase use the same number of helper threads while different phases may use different numbers of helper threads. Table 7.4 summarizes the notations used in the performance model.

data copy time (T_{cp}) in an interval i includes the time to copy data needed by the interval $i + 1$ from PM to DRAM (T_{cp}^{in}), and the time to copy data updated in the interval $i - 1$ from DRAM to PM (T_{cp}^{out}).

$$\begin{aligned}
 T_{cp}^{in}(Thrd_{cp}) &= \frac{data_{in}}{BW^{PM_to_DRAM}(Thrd_{cp})} \\
 T_{cp}^{out}(Thrd_{cp}) &= \frac{data_{out}}{BW^{DRAM_to_PM}(Thrd_{cp})},
 \end{aligned}
 \tag{7.1}$$

where $data_{in}$ and $data_{out}$ are the sizes of data needed to copy in and out of DRAM for an interval; $BW^{PM_to_DRAM}(Thrd_{cp})$ and $BW^{DRAM_to_PM}(Thrd_{cp})$ are the data copy bandwidth in and out of DRAM respectively. These bandwidths are the functions of the number of helper threads ($Thrd_{cp}$). Therefore, T_{cp} is also a function of $Thrd_{cp}$.

Equation 7.1 consider performance difference between copying data from DRAM to PM and from PM to DRAM. In our implementation, copying data in two directions happens in parallel. If memory bandwidth is not a bottleneck, we have

$$T_{cp} = \max(T_{cp}^{in}, T_{cp}^{out}). \quad (7.2)$$

Overlap constraint. Copying data happens in parallel with WarpX execution. The data copy time should be no longer than the WarpX execution time, i.e.,

$$T_{cp}(Thrd_{cp}) \leq T_{comp}(Thrd_{comp}), \quad (7.3)$$

where T_{comp} is the execution time of an interval when the particles accessed by the interval are all in DRAM. T_{comp} is a function of the number of application threads ($Thrd_{comp}$) in an MPI rank.

Bandwidth constraint. The bandwidth consumption due to copying data should not reduce the bandwidth available for WarpX execution. Assume that without copying data, the bandwidth consumption of WarpX execution is BW_{comp} , including both read from and write to PM.

$$\begin{aligned} BW_{comp}(Thrd_{comp}) + BW^{PM_to_DRAM}(Thrd_{cp}) \\ + BW^{DRAM_to_PM}(Thrd_{cp}) \leq BW_{max}/N \end{aligned} \quad (7.4)$$

where BW_{max} is the peak memory bandwidth constrained by the hardware and N is the number of MPI ranks (We assume BW_{max} is evenly partitioned between MPI ranks). BW_{max} needs to satisfy the following equation to prevent performance loss.

$$BW_{max} = \max(BW_{max}^{DRAM_to_PM}, BW_{max}^{PM_to_DRAM}) \quad (7.5)$$

$BW_{max}^{DRAM_to_PM}$ and $BW_{max}^{PM_to_DRAM}$ are the peak memory bandwidth supported by hardware from DRAM to PM and from PM to DRAM, respectively.

Thread constraint. The number of application threads and helper threads should be no larger than the maximum number of threads assigned to an MPI rank ($Thrd_{max}$), i.e.,

$$Thrd_{comp} + Thrd_{cp} \leq Thrd_{max}. \quad (7.6)$$

Optimization goal. Assume that T'_{comp} is the execution time for an interval, given $Thrd_{comp}$ and $Thrd_{cp}$ threads for WarpX execution and copying data, respectively. T'_{comp} is a function of $Thrd_{comp}$ and $Thrd_{cp}$. The goal of our performance modeling is to minimize T'_{comp} (Equation 7.7),

subject to the constraints of overlap (Constraint 7.3), bandwidth (Constraint 7.4) and threads (Constraint 7.6), i.e.,

$$\min(T'_{comp}(Thrd_{comp}, Thrd_{cp})). \quad (7.7)$$

BW_{max} and $Thrd_{max}$ are known from offline profiling; $BW^{DRAM_to_PM}()$ and $BW^{PM_to_DRAM}()$ are measured by a microbenchmark at various numbers of data copy threads; $data_{out}$ and $data_{in}$ are known from $FArrayBox$, whose value is set at the beginning of each iteration. Therefore, based on $data_{out}$ and $data_{in}$, we can calculate T_{cp} using Equation 7.2 given $Thrd_{cp}$.

We build $T_{comp}()$ based on online profiling and empirical observation. In particular, we use an interval in the second iteration of the main computation loop to measure the execution time online, and use $Thrd_{max}$ as $Thrd_{comp}$ during the execution of the interval. This measurement is done after static data placement and after loading the required particles by the interval into DRAM. Furthermore, we empirically observe that the execution of WarpX using various input problems is not bounded by memory bandwidth on Optane (see Section 7.3.1); Using $Thrd_{max}$ as $Thrd_{comp}$ gives the best performance. Using $Thrd_{max} - 1$ and $Thrd_{max} - 2$ as $Thrd_{comp}$ give less than 10% performance loss, while using the number of threads smaller than $Thrd_{max} - 2$ for $Thrd_{comp}$ causes more than 20% loss. Hence, we use the measured online execution time as the result of $T_{comp}(Thrd_{comp})$, when $Thrd_{comp} \in [Thrd_{max} - 2, Thrd_{max}]$. We do not consider other cases of $Thrd_{comp}$ to avoid performance loss of WarpX execution. Note that this approach gives us a high requirement on data copy overhead because of Constraint 7.3.

We employ a similar approach to build $BW_{comp}()$. We measure memory bandwidth in an interval in the second iteration of simulation and using $Thrd_{max}$ as $Thrd_{comp}$. This memory bandwidth is used for $Thrd_{comp} \in [Thrd_{max} - 2, Thrd_{max}]$. We do not consider other cases to avoid performance loss.

We use the following approach to find the optimal $Thrd_{comp}$ and $Thrd_{cp}$ to minimize $T'_{comp}()$. We use Constraints 7.4 and 7.6 to select the numbers of helper threads to meet the bandwidth constraint. Then, among the selected numbers, we use Constraint 7.3 to find those that meet the overlap constraint. Finally, we choose the smallest number as the optimal number of helper thread from those selected numbers. Given the constraints, the WarpX execution time is minimized, $T'_{comp} = T_{comp}(Thrd_{max})$.

Our modeling approach is lightweight, because we avoid exhaustive search of all combinations of $Thrd_{comp}$ and $Thrd_{cp}$ by eliminating those that can obviously cause performance loss. The overhead to find the optimal is almost zero.

7.4.3 Implementation Details

WarpX-PM is implemented as a patch to WarpX and AMReX. Running WarpX with WarpX-PM on Optane (or other HM) requires no efforts from the user. The statistics of modifications given by git diff is 15 files changed, 1031 insertions(+), 12 deletions(-).

WarpX-PM uses pthread to implement helper threads for each MPI rank. For static data placement, data objects that needed to be placed in DRAM are allocated into DRAM NUMA nodes using *numa_alloc_local()*. For dynamic data placement, each MPI process pre-allocates a 500MB temporary space in DRAM to copy particles between DRAM and PM. We use 500MB because the dynamic data placement handles particles batch by batch, and the batch size is determined by `FArraybox`. The size of all particles in one `FArraybox` is bounded by 500MB. All MPI processes evenly partition DRAM initially. To accommodate the size variance of short-lived data objects across iterations, WarpX-PM increases the temporary space by reserving extra 100MB DRAM space for each rank.

Avoiding NUMA effects is important for high performance on an Optane-based machine with multiple sockets, each equipped with both DRAM and PM [155]. We observe that allocating data in remote DRAM and PM nodes (i.e., DRAM and PM on the remote socket) leads to up to 2x performance loss for large input problems in WarpX. To address this NUMA effect, in WarpX-PM, once an MPI rank is pinned to a processor, those DRAM spaces for static and dynamic data placements are allocated from local DRAM NUMA nodes. Also, all data objects of the MPI process are allocated from local PM nodes.

WarpX-PM uses high-performance data copying to implement data placement based on AVX-512 streaming load/store intrinsics and multi-threading. Alternatively, we could use a page migration mechanism such as *move_pages()* and *mmap()* to implement data migration between DRAM and PM instead of data copying. However, these data migration mechanisms work at the page level, requiring setting up a mapping between data objects and pages, which is difficult to implement at the user level. Furthermore, these mechanisms can cause frequent TLB misses because of page remapping, which leads to performance loss [234]. Note that our data copying mechanism in WarpX does not impact program correctness because our implementation has no pointer alias – the pointers pointing to the old data is updated after data copying.

Table 7.5: Input problems used in evaluation

ID	Type	# of cells	# of particles	Peak consumption
A	Laser-driven	(512, 512, 4096)	1.1B	228.5 GiB
B	Laser-driven	(704, 704, 5664)	8.4B	1.2 TiB
C	beam-driven	(512, 512, 4096)	2.1B	306 GiB
D	beam-driven	(864, 864, 7200)	10.7B	960 GiB
E	Uniform-plasma	(384, 384, 3104)	3.7B	525 GiB
F	Uniform-plasma	(512, 512, 4096)	8.6B	1.2 TiB
G	Laser-driven	(256, 256, 2048)	134.2M	19.2 GiB

* The names of particle species of A, E, F and G are set to *electrons*; the names of B are set to *electrons*, *ions* and *beam*; the names of C and D are set to *driver*, *plasma_e*, *plasma_p*, *beam* and *driverback*. The blocking factor is 32.

Table 7.6: Platform Specifications

Processor	2 nd Gen Intel Xeon Scalable processor
Cores	2.4 GHz (3.9 GHz Turbo frequency \times 24 cores (48 HT) \times 2 sockets
L1-icache	private, 32 KB, 8-way set associative, write-back
L1-dcache	private, 32 KB, 8-way set associative, write-back
L2-cache	private, 1MB, 16-way set associative, write-back
L3-cache	shared, 35.75 MB, 11-way set associative, non-inclusive write-back
DRAM	six 16-GB DDR4 DIMMs \times 2 sockets (192 GB in total)
PM	six 128-GB Optane DC NVDIMMs \times 2 sockets (1.5 TB in total)
Interconnect	Intel UPI at 10.4 GT/s, 10.4GT/s, and 9.6 GT/s

7.5 Evaluation

7.5.1 Experimental Setup

Table 7.6 summarizes the hardware features of our testbed. When the Optane DC PMM is in app-direct mode and exposed as NUMA nodes, we use `numactl` [118] to control data placement on PMM and DRAM. The platform runs Fedora 29 (Linux 5.1.0). We use the Intel Processor Counter Monitor (PCM) tool [204] to access hardware counters to collect core activities and off-core events.

Table 7.5 summarizes input problems for evaluation. The input problems come from various plasma accelerator simulations with a wide range of memory consumption (up to 1.2TB). We use WarpX 20.04, OpenMPI 4.0.2 and GCC 7.5.0. For all problems except Problem G (a relatively small input), we run 10 iterations and report average execution time per iteration. There is less than 1% difference in average execution time if we use more than 10 iterations. For Problem G, we run 30 iterations to report average execution time, because average execution time becomes stable only after 20 iterations.

7.5.2 Evaluation Results

Overall performance. We compare WarpX-PM with Optane-only (i.e., no DRAM) and two common strategies (i.e., NUMA first-touch and memory mode) to use Optane-based systems. We evaluate Problems A-F in Table 7.5. All these problems have peak memory consumption larger than DRAM (192 GB). For the small Problem G, all data objects can be placed in DRAM. Hence, there is almost no performance difference between NUMA first-touch, memory mode, and WarpX-PM.

Figure 7.4 reveals that WarpX-PM performs the best in all cases. On average, WarpX-PM outperforms memory mode, Optane-only, and NUMA first-touch by 34.4%, 64.6%, and 41%, respectively. We notice that NUMA first-touch performs worse than memory mode and WarpX-PM. NUMA first-touch decides data placement based on when data allocation happens, instead of memory access patterns, which leads to sub-optimal data placement if a performance-critical data object is allocated at a later stage of execution. For example, particles are allocated before fields in WarpX, and NUMA first-touch places particles in DRAM, which forces fields to go to PM because of limited DRAM capacity. However, fields is more frequently accessed throughout simulation — placing it into PM leads to substantial performance loss. WarpX-PM avoids this problem because it prioritizes the placement of fields over particles on DRAM.

WarpX-PM outperforms memory mode because it avoids DRAM-cache thrashing for small and short data objects. DRAM-cache thrashing happens because of memory accesses to the large data object, particles. Without application knowledge, the DRAM cache may evict small and short-lived data objects to make space for particles.

WarpX execution has large performance variance in runs of Problem A in memory mode. This problem has peak memory consumption only slightly larger than DRAM. The performance variance is up to 30.5%. Such performance variance in memory node has been confirmed by Intel, and imposes a big challenge on controlling performance variability in HPC applications. WarpX-PM avoids this performance variance problem because of its static placement of critical data objects.

Performance breakdown. We quantify the contributions of the static data placement and dynamic data placement techniques in WarpX-PM, to the performance improvement on each execution phase. We further compare WarpX-PM with memory mode (the second best in Figure 7.4) to demonstrate the effectiveness of WarpX-PM. Figure 7.5 presents the results.

For input problems with large memory footprint (e.g., Problems B and D with 1.2TB peak memory consumption), static data placement outperforms memory mode by 29% on average.

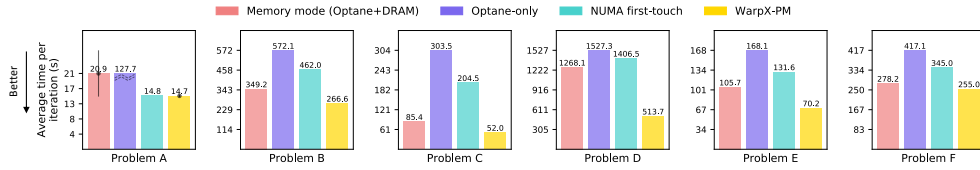


Figure 7.4: Performance comparison between Memory mode, Optane-only, NUMA first-touch and WarpX-PM.

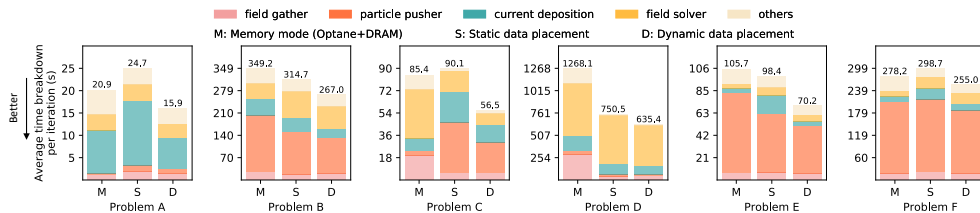


Figure 7.5: Performance breakdown of main phases of execution to compare the static and dynamic placement with memory mode.

Memory mode cannot work well for the large input problems, because metadata and temporary data are not efficiently cached into DRAM. Furthermore, static data placement effectively reduces the execution time on `field gather` and `others` (compared to memory mode) by 10% and 7% on average over all input problems. `field gather` and `others` involve large amount of metadata and temporary data objects access. Static data placement effectively prevents data migration for these two phases and thus avoids the migration overhead.

Dynamic data placement improves the performance of `particle pusher`, `current deposition` and `field solver` by 11%, 17%, and 12%, compared with static data placement. By proactively prefetching particles from PM to DRAM, dynamic data placement outperforms memory mode and static data placement by 34% and 41%.

Comparison with state-of-the-art. We compare WarpX-PM with a state-of-the-art page migration system for HM, named *improved active list* (IAL) [234]. This system improves an existing page replacement mechanism in the Linux kernel (i.e., an FIFO-based active list). Among the 7 input problems listed in Table 7.5, we can only run three of them successfully with IAL. Running other problems with IAL suffers from either extremely poor performance (10x worse than WarpX-PM) or segmentation faults. Figure 7.6 shows the results.

Figure 7.6 reveals that WarpX-PM outperforms IAL by 83.3% on average and up to 96.6%. There are three main reasons for the inferior performance of IAL. First, IAL is a reactive approach – it takes effects only after it collects enough information on memory accesses. This indicates that

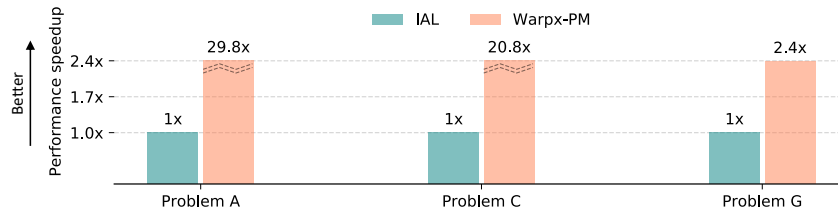


Figure 7.6: Performance comparison between IAL (a state-of-the-art page migration solution for HM) and WarpX-PM.

it cannot efficiently prefetch data objects into DRAM to reduce data movement cost. Second, IAL periodically samples memory page accesses to identify page hotness. Finding hot pages from a large amount memory pages (tens of millions) incurs significant overhead. Third, IAL heavily relies on helper threads to enable parallel page migration for high performance. However, IAL does not consider the impact of using helper threads on the WarpX execution. Using an excessive number of helper threads decreases computation capability available for WarpX and consumes large memory bandwidth, which negatively impact the WarpX performance.

Memory bandwidth analysis. Figure 7.7 depicts read/write bandwidth for memory mode and WarpX-PM. We use input Problem F, because its peak memory consumption is the largest and pressures the memory bandwidth. Compared with memory mode, WarpX-PM consumes higher DRAM bandwidth, indicating that fast memory accesses happen more often in WarpX-PM to make best use of DRAM. More specifically, for execution phases that only involve static data migration (i.e., `field gather` and `others`), PM bandwidth consumption is lower than memory mode, indicating the effectiveness of static data placement. For execution phases that involve dynamic data migration (`current deposition`, `field solver` and `particle pusher`), WarpX-PM has higher PM bandwidth than memory mode. This is because dynamic data placement prefetches data objects before they are accessed, but data prefetching overhead is hidden by overlapping with the computation.

NUMA effects. Optane-based systems have multiple sockets, each with DRAM and PM DIMMs. Efficient data placement is not only about using DRAM or PM but also about avoiding memory accesses to a remote socket. We compare memory mode, NUMA first-touch with WarpX-PM to quantify NUMA effect by tracking memory traffic between two sockets. We use six input problems whose peak memory consumption is larger than DRAM to allow us to evaluate the NUMA effect fully. Table 7.7 shows the results.

The results show that WarpX-PM has the lowest inter-socket traffic (close to zero). Memory mode is not NUMA-aware and cannot cache accesses to remote PM in a local DRAM [62]. The

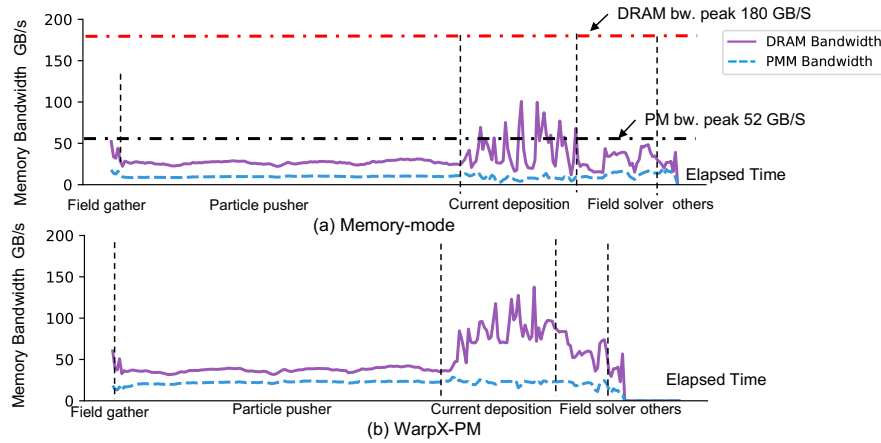


Figure 7.7: Memory bandwidth consumption in one iteration.

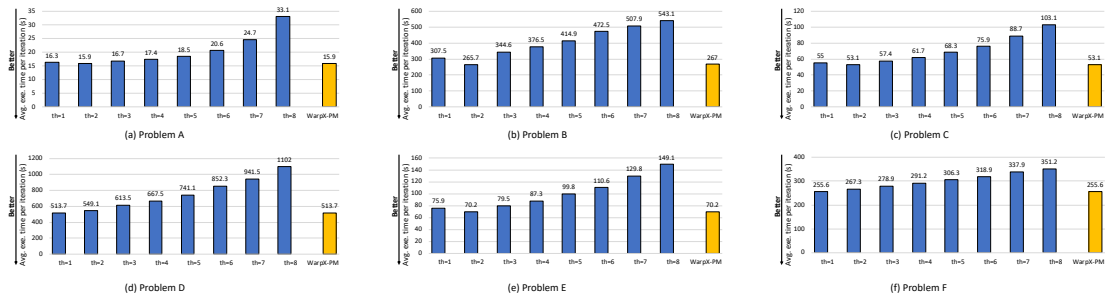


Figure 7.8: Performance with different number of helper threads. “th” is the number of helper threads.

Table 7.7: Quantifying memory traffic between two NUMA nodes.

Problem ID	Remote DRAM traffic (GB)		
	Memory mode	First-touch	WarpX-PM
A	0.92	1.01	0.01
B	4.08	3.75	0.02
C	3.69	4.76	0.02
D	18.09	23.90	0.05
E	1.01	0.90	0.01
F	2.22	2.28	0.01

NUMA first touch policy is NUMA-aware, but data may be distributed to remote DRAM when the local DRAM is exhausted. WarpX-PM avoids these problems by explicitly placing data in local buffers (Section 7.4.3).

Effects of performance modeling. We evaluate the effectiveness of the performance model in determining the number of helper threads. The performance variance due to different numbers of helper threads is the same across phases. Hence, we use the same number of helper threads for all phases for evaluation. We manually sweep the number of helper threads and compare their performance with the automatically adapted performance in WarpX-PM. Figure 7.8 shows

the results. For Problem A, B, C, and E, the optimal number of helper threads is two. For Problem D and F, the optimal number of helper thread becomes one. WarpX-PM achieves similar performance as the optimal one, demonstrating the effectiveness of performance modeling. We also notice more than 30% performance loss when the number of helper threads is larger than two. As the number of helper threads increases, the available processor cores for the computation in WarpX simulation decreases, which prolongs the total execution time.

7.6 Related Work

HPC workloads Many works have explored PM-based HM for HPC [56, 140, 154, 226, 227]. Nguyen et. al [140] introduce a multi-version octree on PM to enable adaptive mesh simulation on PM. Unimem [226] uses performance modeling to decide data placement for MPI-based HPC applications. Siena [154] explores rich organizations and configurations of HM architecture for HPC applications to determine optimal system designs. Tahoe [227] combines a machine learning model and an analytical model to predict application performance across multiple memory components for task-parallel programs. NVStream [56] uses non-temporal store and delta compression to reduce overhead for maintaining crash consistency and reduce I/O traffic for HPC workloads. These works use emulated PM to demonstrate their functionality. Recent works also characterizing HPC applications on Optane [152, 157, 214, 222]. Instead, our work focuses on performance analysis and optimization of a production-level code (WarpX) for realistic simulations on real PM hardware.

Database and graph workloads. Recent works also propose various performance optimizations of databases and graph workloads on the Optane PM [39, 62, 101, 107, 235]. Yang et. al [235] analyze the Optane architecture to optimize database and file system. TimeStone [101] solves the problem of poor scalability of durable transaction memory (DTM) on Optane by adopting multi-version concurrency control and a DRAM buffer. RECIPE [107] converts concurrent DRAM indexes to crash-consistent indexes on Optane. Gill et. al [62] evaluate four graph analytics frameworks and optimize performance by mitigating the NUMA effect of Optane. ATMem [39] employs a sampling-based profiler to select performance-critical data regions in graph applications on Optane.

7.7 Conclusions

The emerging large-capacity PM enables high-resolution large-scale scientific simulations.

However, leveraging PM for production-level HPC codes on realistic problems remains to be investigated. In this paper, we focus on WarpX, a mission-critical plasma simulation code, as a use case to study PM implications on its performance. We demonstrate the PM benefits in simulation scales and propose a set of performance optimization strategies driven by detailed performance analysis. We improved the WarpX execution on Optane-only by 64.6% and outperformed DRAM-cached, the NUMA first-touch policy, and a state-of-the-art HM solution by 34.4%, 41% and 83.3%, respectively.

Chapter 8

Scalable Page Management for Multi-Tiered Large Memory Systems

Multi-terabyte large memory systems are emerging. They are often characterized with more than two memory tiers for large memory capacity and high performance. Those tiers include slow and fast memories with different latencies and bandwidths. Making effective, transparent use of the multi-tiered large memory system requires a page management system, based on which the application can make the best use of fast memories for high performance and slow memories for large capacity. However, applying existing solutions to multi-tiered large memory systems has a fundamental limitation because of non-scalable, low-quality memory profiling mechanisms and unawareness of rich memory tiers in page migration policies. We develop HM-Keeper, an application-transparent page management system that supports the efficient use of multi-tiered large memory. HM-Keeper is based on two design principles: (1) The memory profiling mechanism must be adaptive based on spatial and temporal variation of memory access patterns. (2) The page migration must employ a holistic design principle, such that any slow memory tier has equal opportunities to directly use the fastest memory. We evaluate HM-Keeper using common big-data applications with large working sets (hundreds of GB to one TB). HM-Keeper largely outperforms seven existing solutions by 15%-78%.

8.1 Introduction

Memory hierarchy has continued deepening to cope with ever-increasing demands from applications. Multi-tier memory systems that started from multi-socket non-uniform memory

access (NUMA) architecture is now a de-facto solution for building scalable and cost-effective memory systems. For instance, the Amazon EC2 High Memory Instance has three DRAM-based memory tiers built upon eight NUMA nodes, providing up to 12 TB memory [82]. Recently, the commercial availability of new memory devices, such as high-bandwidth memory (HBM) and high-density persistent memory (PM), started expanding a new dimension of multi-tier memory systems. A multi-tier memory system implemented with heterogeneous memories and NUMA architecture can easily exceed two memory tiers. Top tiers typically feature lower memory latency or higher bandwidth but smaller capacity, while bottom tiers feature high capacity but lower bandwidth and longer latency. When high-density PM is in use, e.g., Intel’s Optane DC persistent memory [201], a multi-tier large memory system could enable terabyte-scale graph analysis [44, 62, 160], in-memory database services [20, 36, 235], and scientific simulations [140, 231] on a single machine.

Muti-tier memory systems can be managed transparently by hardware, e.g., Intel’s Optane can configure DRAM as a direct-mapped cache for PM. Such hardware supports require no application modifications or OS changes, and naturally, are often the first option to be explored on a new system. However, cache-like mechanisms rely on good data locality to gain performance. Indeed, applications with low data locality have shown unsatisfactory performance on such systems due to the extra overhead in managing cache [155] and write amplification [21]. Another shortcoming of cache mechanisms is the loss of sizable memory capacity. Unlike processor caches, which are often in MBs, PM-based memory systems, e.g., Intel Cascade Lake processor, could contain hundreds of GB capacity in DRAM tiers.

Software-based page management can leverage multi-tier large memory systems more efficiently than hardware mechanisms because it can gain deeper insights into memory access patterns. Most of software solutions [14, 84, 85, 98, 119] consist of three components – a profiling mechanism, a migration policy, and a migration mechanism. A profiling mechanism is critical for identifying performance-critical data in applications and is often realized through tracking page accesses. A migration policy chooses candidate pages to be moved to top tiers. Finally, the effectiveness of a page management solution directly depends on whether its migration mechanism can move pages across tiers at low overhead.

Problems in profiling. Existing memory profiling mechanisms [78, 95, 104, 248] manipulate specific bits in page table entries (PTEs) to track memory accesses at a per-page granularity. The profiling overhead scales linearly with the number of tracked pages. Our evaluation shows that tracking millions of pages could take several seconds – too slow to respond to time-

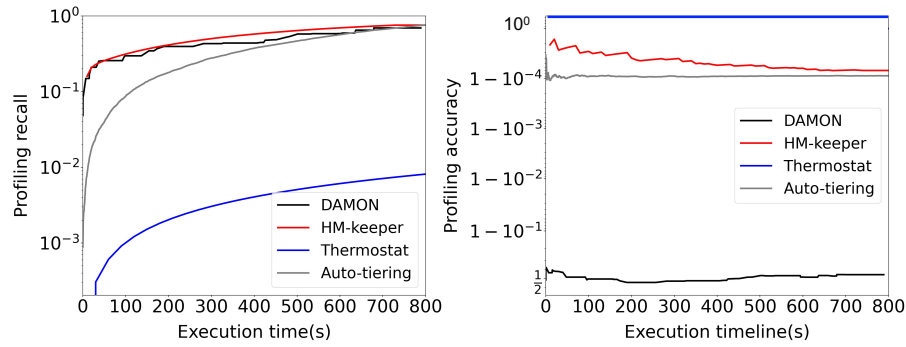


Figure 8.1: Comparison of different memory profiling methods in terms of their effectiveness of identifying frequently accessed pages (hot pages). The profiling overhead is set as 5% of total execution time.

changing access patterns, and causes 20% slowdown in TPC-C against VoltDB [213]. Some solutions [14, 84, 85, 119, 151, 189] only profile a small set of randomly-chosen pages based on PTE manipulation or performance counters, or heavily rely on the user to configure the profiling method to reduce profiling overhead. However, such a strategy compromises profiling quality and may miss frequently-accessed pages and time-changing access patterns.

Figure 8.1 compares multiple profiling methods in Thermostat [14], AutoTiering [98], DAMON [151, 189], and our method. These profiling methods represent state-of-the-art. We use the GUPS [67] benchmark with 512GB working set and have priori knowledge on which pages are accessed at least twice in a profiling interval (i.e., hot pages). In this workload, hot pages remain stable throughout the execution. We report profiling recall (i.e., the ratio of the number of correctly detected hot pages to the number of hot pages identified by priori knowledge) and profiling precision (i.e., the ratio of the number of correctly detected hot pages to the number of total detected hot pages including mis-identified hot pages). With the same profiling overhead (5%), Thermostat and AutoTiering take long time to identify hot pages (see Figure 8.1). DAMON takes shorter time but about 50% of hot pages detected by DAMON are actually not hot. Because of low profiling quality, DAMON, Thermostat, and AutoTiering perform at least 15% worse than our method. In general, there is a lack of scalable and high-quality profiling mechanism for large memory systems.

Problems in migration. Existing solutions to multi-tiered memory are built upon an abstraction extended from traditional memory hierarchy, where page migration occurs between two neighboring tiers. However, such abstraction could limit multi-tiered memory systems. First, migrating pages from the lowest to the top tier, at tier-by-tier steps, takes time, e.g., tens of microseconds for a 4K page. Also, frequent data movement across tiers is needed to accommodate

time-changing access patterns, different from occasional swapping between memory and storage.

To leverage a multi-tier large memory system efficiently, we argue that the following four principles must be upheld.

- *Scalable quality-aware profiling.* Formal metrics that quantify performance impact from pages need to be established to guide page selection for profiling. By only tracking the most performance-critical pages at a time, a solution can guarantee adaptiveness to time-changing patterns and controlled overhead.
- *Global view of memory regions in all tiers.* Only tracking data in the fastest tier is insufficient because large memory regions are forced to reside in lower tiers even though they may contain small but performance-critical subregions. This is particularly true on terabyte-scale systems where data in the top tiers can only accommodate a small portion of total working sets and thus insufficient to provide a comprehensive view of system-wide data. Thus, a global view of all memory regions in all tiers is crucial.
- *Tier-bypassing migration.* Emerging large-memory systems feature multiple tiers. Migration through neighboring tiers is too slow. Instead, performance-critical pages should be promoted to the top tier bypassing intermediate tiers, i.e., pages in all slower tiers have equal chances to be promoted to the fastest tier. Victim pages evicted from the top tier should be progressively demoted into the next available lower tier instead of the bottom tier as in the existing swapping-based solution in Linux [119] (i.e., AutoNUMA) because these pages are still likely to be accessed.
- *Pattern-aware migration mechanism.* The page migration mechanism includes multiple stages. Parallelism between stages is often ignored. Considering read/write patterns of page accesses, such parallelism can be exposed, which is critical to improve page migration performance on multi-tiered memory with frequent page migration.

In this work, we contribute a software-based page management solution called HM-Keeper that realizes the four principles on terabyte-scale multi-tier memory.

HM-Keeper maintains a flat global view of all memory regions in all memory tiers. Profiling quality and overhead are distributed proportionally according to a memory region's importance. Memory regions with a higher impact on performance are tracked with higher fidelity, i.e., more pages are selected for profiling. In particular, HM-Keeper uses spatial and temporal variation as the key metrics to quantify the performance impact of a memory region. Over time, HM-Keeper adaptively merges and splits memory regions based on their similarity in spatial and temporal locality so that pages in a memory region have similar access patterns.

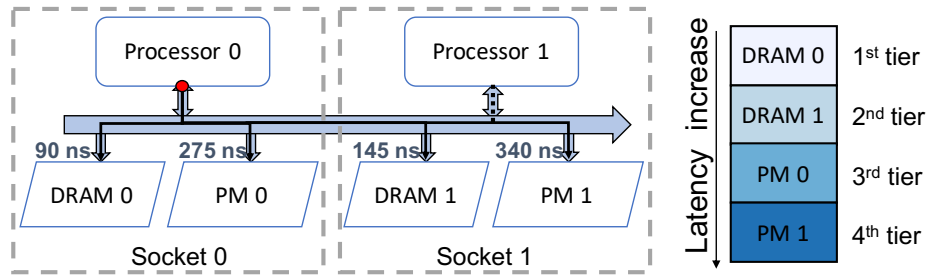


Figure 8.2: An example of multi-tiered memory system.

HM-Keeper uses the “fast promotion slow demotion” policy for page migration. Hot pages identified in all lower tiers can be directly promoted to the top tier, minimizing data movement through tiers. When a page is migrated out of the top tier to accommodate more important pages, the page will be moved to the next lower tier with available space. HM-Keeper dynamically chooses from an asynchronous page copy-based scheme and a synchronous page migration scheme, based on the read/write pattern of the migrated page, to minimize migration time.

Evaluation. We rigorously evaluated HM-Keeper against seven state-of-the-art solutions, including two industry-quality software solutions (Intel Memory-optimizer [84] and Intel Memory-tiering [85]), two state-of-the-art solutions (AutoTiering [98] and HeMem [176]), an existing solution in Linux (AutoNUMA [41]), a hardware-based solution (Optane Memory Mode), and first-touch NUMA. HM-Keeper is also compared against two kernel-based page migration solutions (the ones in Linux and Nimble [234]). HM-Keeper outperforms Intel memory-optimizer, Intel memory-tiering, AutoTiering, AutoNUMA, Memory Mode, first-touch NUMA and HeMem by 32%, 22%, 20%, 25%, 24%, 20%, and 24% on average. HM-Keeper outperforms the Linux and Nimble migration approaches by 40% and 36% for read-intensive workloads, and has similar performance for write-intensive workloads.

8.2 Background and Related Work

8.2.1 Multi-Tiered Large Memory Systems

In this section, we introduce memory organization in typical multi-tiered large-memory systems. Note that recent works on using network-attached memory [132, 186, 191, 206, 217], i.e., *cross-machine* multi-tiered systems are beyond the scope of this work. Figure 8.2 shows an example of the Intel Optane-based large memory system with two sockets and four memory components (i.e., DRAM 0-1 and PM 0-1). Each memory component is called a *tier* and has

different memory latency.

8.2.2 Large Memory Systems

Given the emergence of large memory systems, there is a pressing need of studying effectiveness and scalability of system software and hardware to support them. We review the related works as follows.

Software support for page management. Recent work manages large memory systems based on an existing NUMA balancing solution in Linux, AutoNUMA [119]. AutoNUMA randomly profiles 256MB memory pages to reduce memory profiling overhead. AutoNUMA migrates pages to reduce memory accesses across NUMA nodes, and does not consider page hotness. Intel Memory-tiering [85] uses the same profiling method as AutoNUMA but adds extension to support multi-tiered large memory. It balances memory accesses between sockets first, and then balance memory accesses across memory tiers within each socket. As a result, a hot page takes a long time to migrate to the fastest memory for high performance. Intel Memory-optimizer [84] randomly samples pages for profiling, and only migrates pages between two memory tiers within the same socket, failing to exploit fast memory across sockets. AutoTiering [98] is a state-of-the-art solution. It uses random sampling as AutoNUMA, and introduces flexible page migration between memory tiers. However, it does not have a systematic migration strategy guided by page hotness. HeMem [176] is a state-of-the-art solution for two-tiered PM-based HM. HeMem leverages sampling-based hardware performance counters to identify hot pages and fails to explore more than two tiers.

Mitosis [12] and vMitosis [149] explore how to efficiently place page tables across sockets in large memory systems. They complement HM-Keeper, because HM-Keeper focuses on application-level data (the most memory-consuming data), not kernel-level objects. \emptyset sim [129] recognizes another pressing problem in large memory systems: how to enable rapid, early prototyping and exploration of system software for large memory. \emptyset sim harnesses the fact that many workloads follow the same control flow regardless of their input to make huge simulations feasible and fast via memory compression. HM-Keeper can use \emptyset sim for fast evaluation.

Hardware-managed memory caching. Some large memory systems use fast memory as a hardware-managed cache to slow memory. For example, in Intel's Optane PM, DRAM can work as a hardware-managed cache to persistent memory in the *Memory Mode*. However, this solution results in data duplication in fast and slow memories, wasting fast memory capacity. It also causes serious write amplification when there are memory cache misses. Recent work [21]

reveals that Memory Mode causes more than 3x extra writes and 50% bandwidth drop, compared with software-based solutions.

8.2.3 Two-Tiered Heterogeneous Memory

Heterogeneous memory (HM) combines the best properties of memory technologies optimized for latency, bandwidth, capacity, and cost, but complicate memory management. There are OS-level, application-transparent solutions that measure data reuse and migrate data for performance [14, 48, 49, 78, 95, 96, 104, 123, 176, 234]. However, they can cause large and uncontrolled profiling overhead or low profiling quality, and are not designed for more than two memory tiers.

There are application-specific solutions that leverage application domain knowledge to reduce profiling overhead, prefetch pages from slow memory to fast memory, and avoid slow-memory accesses. Those solutions include big data analysis frameworks (e.g., Spark [216]), machine learning applications [77, 178, 181], scientific computing [140, 154, 231], and graph analysis [44, 62, 160]. These solutions show better performance than the application-transparent, system-level solutions, but require extensive domain knowledge and application modifications. Instead, HM-Keeper is an application-transparent solution.

8.3 Overview

HM-Keeper consists of three components: (1) an adaptive memory profiling mechanism that achieves high quality at low overhead; (2) a page-migration strategy that leverages a global view of all memory regions in all tiers to make informed decisions; and (3) a page-migration mechanism that adapts data copy schemes based on page access patterns.

HM-Keeper partitions the virtual address space into memory regions. It periodically profiles memory accesses and migrates pages. In each profiling interval, HM-Keeper samples one or more pages per memory region. This region-based profiling strategy captures spatial locality in each region. Memory regions can be dynamically merged or split under the guidance of quantitative analysis on the profiling overhead. Such adjustments provide opportunities to redistribute sampling quotas between memory regions under a fixed profiling overhead to improve profiling quality.

HM-Keeper uses a holistic approach to decide page migration between memory tiers. By calculating the exponential moving average of page hotness collected from all profiling intervals, HM-Keeper learns the distribution of hot memory regions in *all* memory tiers. Guided by this

information, HM-Keeper promotes hot pages from any memory tier to the fastest tier without going through any intermediate layers. It demotes pages tier by tier when there is not enough space in the fast memory tiers.

When migrating pages, we introduce an asynchronous page-copy mechanism that overlaps page copying with application execution. This mechanism reduces the overhead of page copy. However, the asynchronous page copy can come with the time cost of *extra* page copy, because when a page is updated during copying, the page has to be copied again. The traditional, synchronous page-copy mechanism does not need extra page copy, but completely exposes the overhead of page copy into the critical path. Hence, HM-Keeper uses a hybrid approach that takes advantage of both asynchronous and synchronous mechanisms. HM-Keeper selects migration mechanism based on whether page modification happens during migration.

8.4 Adaptive Memory Profiling

Figure 8.3 depicts the profiling workflow in HM-Keeper. The fundamental mechanism tracks page accesses by utilizing PTE reserved bits and PTE scan. In particular, each PTE maintains an access bit, which indicates the access status of the corresponding page. The access bit is initially set to 0, but changed to 1 by the memory management unit (MMU) when the corresponding page is accessed. By repeatedly scanning PTE to check the value of the access bit and resetting the access bit to 0 if it is found to be 1, page accesses can be monitored. This mechanism is commonly used in existing works [78, 151]. However, using it naively would impose high overhead on large memory systems.

Scanning all PTEs to track memory accesses of each page for large memory is expensive. For example, scanning a five-level page table for 1.5 TB memory with page size of 2MB in an Optane-based platform (see Table 8.1 for hardware details) with one helper thread takes more than one second, which is too long to capture varying workload behaviors. To avoid such a long profiling time, it is natural to sample pages in the address space for profiling. In a large memory system, sampling pages is challenging, because there are a large number of pages to choose for sampling, and the profiling quality with unguided, random sampling [14, 84, 85, 98, 151] can lead to poor performance as discussed in Section 8.1. We introduce an adaptive memory profiling method to improve profiling quality while dynamically enforcing limits on profiling overhead.

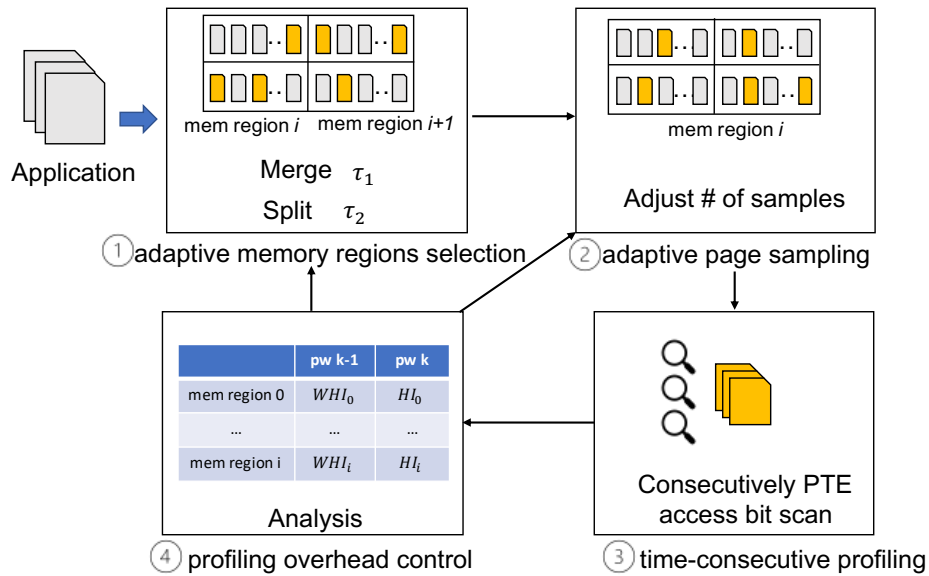


Figure 8.3: The overview of memory profiling in HM-Keeper.

8.4.1 Adaptive Memory Regions

HM-Keeper partitions the virtual address space of a process into memory regions for profiling. By default, a memory region is a contiguous address space mapped by a last-level page directory entry (PDE). This indicates that in a typical five-level page table, the memory region size is 2MB by default. During program execution, whenever a last-level PDE is set as valid by OS, the corresponding memory region is subject to profiling. Memory regions can be dynamically merged to reduce profiling overhead, or split to improve profiling quality. Therefore, different memory regions can have different sizes.

We use the last-level PDE to decide the initial size of each memory region, based on experimental analysis. If a higher-level PDE is used to decide the initial size of memory regions, the default memory region becomes at least 1GB. With such a large memory region, data objects with different memory access patterns are likely to reside in the same memory region [178, 244]. Since memory region is the basic unit for memory profiling and migration, those data objects will be migrated together, even though they may favor different memory tiers.

In each memory region, HM-Keeper samples one or more 4KB pages for profiling (discussed in detail in Section 8.4.2). HM-Keeper scans access bits in PTEs multiple times in a profiling interval.

Multiple scans of PTEs. In a profiling interval, the access bit in a PTE corresponding to a page sample is scanned multiple times. The total number of scans per PTE per profiling interval

is subject to a constraint, num_scans .

We use the above multi-scan method, instead of single-scan in the existing work [78, 119, 151] to reduce skewness of profiling results. A page in a profiling interval can be accessed multiple times. In a profiling interval, a single-scan method can only detect whether a page is accessed or not, but cannot accurately capture the number of memory accesses. Although aggregating memory accesses across multiple profiling intervals could alleviate this problem, the skewness of profiling results will be accumulated over time (see Section 8.5), leading to sub-optimal migration decisions. Using the multi-scan method avoids this problem.

At the end of a profiling interval, the average number of accesses to all sampled pages in a memory region is used as the *hotness indication* of that memory region. Based on the hotness indication, HM-Keeper may either merge or split memory regions.

Merge memory regions. HM-Keeper actively looks for opportunities to merge contiguous memory regions at the end of a profiling interval. Two contiguous regions are merged, if their difference in the hotness indication collected in the most recent profiling interval is smaller than a threshold τ_1 .

Split a memory region. HM-Keeper also checks whether a memory region should be split to ensure pages in the same region have similar hotness in each interval. When the maximum difference in the number of memory accesses among sampled pages in a region is larger than a threshold τ_2 , the memory region is split into two equally-sized ones.

Selection of τ_1 and τ_2 . τ_1 and τ_2 define the minimum and maximum differences in the number of memory accesses among page samples in a memory region. τ_1 and τ_2 fall into $[0, num_scans]$. To avoid frequent merging/splitting and balance between them, τ_1 and τ_2 evenly split the range of $[0, num_scans]$, i.e., $\tau_1 = 1/3 * num_scans$ and $\tau_2 = 2/3 * num_scans$ by default. τ_1 can be dynamically fine-tuned to enforce the limit on profiling overhead, discussed in Section 8.4.3.

8.4.2 Adaptive Page Sampling

Initial page sampling. Initially, each memory region has only one page sample for profiling. Our method for choosing the initial page sample in a memory region is based on a method in Thermostat [14]. In particular, HM-Keeper monitors all pages in the memory region in a profiling interval, identifies those pages with non-zero accesses, and randomly chooses one among those pages. After the initial page sampling, the number of page samples in a memory region can be dynamically changed to improve profiling quality.

After merging of two memory regions, the total number of page samples in the two regions is reduced by half under the constraint that the new memory region should have at least one page sample. This reduction of page samples saves the profiling overhead for the two regions, and allows other memory regions to have more samples without exceeding the overhead constraint.

The saved page-sample quota after merging memory regions is re-distributed to other memory regions. First, HM-Keeper distributes sample quota to the memory regions whose hotness indication shows the largest variance in the last two profiling intervals among all memory regions. Having a large variance of hotness indication in two profiling intervals indicates that the memory access pattern is changing. Adding more page samples for profiling in this case is useful to improve profiling quality.

To efficiently find memory regions with the largest variance of hotness indication among all memory regions, HM-Keeper keeps track of top-five largest variances and the corresponding memory regions when analyzing profiling results. Whenever a new profiling result for a memory region is available, HM-Keeper checks the top-five records and updates them if needed. After the merging, the saved page-sample quota is re-distributed to those top-five memory regions.

After splitting a memory region into two new regions, the page-sample quota in the original region is evenly split between the two new regions. Therefore, splitting does not change the number of total samples. Nevertheless, splitting the memory region brings two benefits. First, the hotness indication, which is the *average* number of accesses to all sampled pages in a memory region, provides better indication of memory accesses to the new, smaller memory regions, hence providing better guidance on page migration. Second, migration is more effective, because using the smaller memory region avoids unnecessary data movement coming with the larger region.

8.4.3 Profiling Overhead Control

We discuss how the profiling overhead control is integrated into adaptive memory regions and page sampling in this section. HM-Keeper supports the user to define a profiling overhead constraint. HM-Keeper respects this overhead constraint while maximizing profiling quality, by dynamically changing the number of memory regions and distributing page-sample quotas between the regions. The overhead constraint is a percentage of program execution time without profiling and migration. For example, in our evaluation section, this overhead constraint is 5%.

Given the length of a profiling interval (t_{mi}), profiling overhead constraint, overhead of scanning one PTE ($one_scan_overhead$), and number of scans per PTE (num_scans), the total number of page samples in all

memory regions that can be profiled in a profiling interval, denoted as num_ps , is calculated in Equation 8.1.

$$num_ps = \frac{t_{mi} \times profiling_overhead_constraint}{one_scan_overhead \times num_scans} \quad (8.1)$$

t_{mi} can be set by the user, as in existing works [14, 78, 151]. $one_scan_overhead$ is measured offline by repeatedly scanning PTEs and then measuring the average scanning time. As HM-Keeper merges or splits memory regions, the total number of page samples in all memory regions remains equal to num_ps to respect the profiling overhead constraint.

The total number of memory regions needs to be smaller than num_ps so that each memory region has at least one page sample. When the total number of memory regions is too large, HM-Keeper dynamically fine-tunes τ_1 (the threshold to merge regions) to merge memory regions more aggressively. τ_1 is gradually increased across profiling intervals, until the number of memory regions is no larger than num_ps .

We do not change num_scans (i.e., the number of scans per page sample) to enforce the profiling overhead constraint, because of its significant impact on profiling quality. Changing num_scans leads to a change of profiling results in all memory regions. For example, in our evaluation, when changing num_scans from 2 to 3, HM-Keeper changes the migration decision for at least 20% of memory regions. We set num_scans as a constant. Our empirical study shows that using a value larger than three leads to no obvious change (less than 5%) in the migration decision.

Memory consumption overhead in HM-Keeper. For each memory region, HM-Keeper stores the hotness indication as an integer. Given a terabyte-scale memory, this memory consumption overhead is only hundreds of MBs. For example, in our Optane-based platform with 1.5 TB memory, the memory overhead to store profiling results is no larger than 600MB, which is small on large memory systems.

8.5 Holistic Page Management

In this section, we discuss the page migration strategy. Essentially, this strategy answers two questions: (1) which memory regions to migrate, and (2) given a memory region to migrate, which memory tier to migrate. HM-Keeper answers the first question by selecting memory regions based on analyzing time-consecutive profiling results, and the second question by using the strategy of “fast promotion but slow demotion”.

8.5.1 Which Memory Region to Migrate?

At the end of each profiling interval, HM-Keeper migrates (or promotes) some memory regions to the fastest memory, and the total size of those memory regions is a constant N ($N=200\text{MB}$ in our evaluation). This is similar to the existing works [84, 85, 98, 132] that periodically migrates a fixed number of pages. If free memory space in the fastest memory is not large enough for migration, some pages in the fastest memory are demoted to the slower memory tiers (see Section 8.5.2).

Select memory regions for promotion. The goal of region promotion is to place the most frequently accessed pages into the fastest memory. HM-Keeper decides which regions to promote based on hotness indication collected from *all* memory regions, regardless which tiers those memory regions are currently in. Hence, the migration decision is holistic. The memory regions with the largest hotness indication are promoted.

HM-Keeper uses time-consecutive profiling results to select regions for promotion. Particularly, HM-Keeper uses the hotness indication collected from the most recent profiling interval *and* the prior profiling intervals. As a result, HM-Keeper captures temporal locality, and avoids page migration due to the bursty memory access pattern in one profiling interval.

HM-Keeper gets time-consecutive profiling results based on the exponential moving average (EMA) of hotness indication collected from all profiling intervals. Given a sequence of data points, EMA places a greater weight and significance on the most recent data points. We define EMA of hotness indication as follows. Assume that HI_i is the hotness indication collected at the profiling interval i for a memory region, the EMA of the hotness indication for that memory region at the profiling interval i , denoted as WHI_i , is defined in Equation 8.2. This equation is a recursive formulation including WHI_i and WHI_{i-1} from the prior interval $i - 1$.

$$WHI_i = \alpha \times HI_i + (1 - \alpha) \times WHI_{i-1} \quad (8.2)$$

α in the above formulation indicates how important the history information is for making the decision. In practice, we set α as 0.5. There are two benefits of using EMA. First, the memory consumption is small. There is no need to store all prior profiling results. Second, the computation of EMA is lightweight, and hence the runtime overhead is low.

Based on the EMA of hotness indication, HM-Keeper uses the following method to select memory regions to migrate. HM-Keeper builds a histogram to get the distribution of EMA of all memory regions. The histogram buckets the range of EMA values, and counts how many and what memory regions fall into each bucket. Given the size of pages to migrate to the fastest memory,

HM-Keeper chooses those memory regions falling into the highest buckets in the histogram to migrate. Building and maintaining the histogram is not costly: Whenever the EMA of hotness indication of a memory region is available, the histogram only needs to be slightly updated accordingly.

8.5.2 Where to Migrate Memory Regions?

Where to promote memory regions? As discussed in Section 8.5.1, memory regions are promoted to the fastest memory tier based on the histogram. It is likely that after a profiling interval, there is no memory region to promote to the fastest memory because those memory regions falling into the highest buckets of the histogram are already there. In that case, memory regions in the lower buckets of the histogram are selected to promote to the second-fastest memory tier, and the accumulated size of those memory regions to migrate should always be N (the total size of memory regions to migrate). In general, HM-Keeper makes the best efforts to promote frequently accessed memory regions to high-performance memory tiers.

Where to demote memory regions? When a memory tier is a destination of memory promotion but does not have enough space to accommodate memory promotion, memory regions in that memory tier are migrated (or demoted) to the next lower memory tier with enough memory capacity. Memory regions for demotion are selected based on the histogram – Memory regions that are in the lowest buckets of the histogram are demoted to the next lower tier. We use the above slow-demotion strategy to avoid performance loss caused by migrating pages that are still likely to be accessed in a low memory tier in near future.

8.6 Adaptive Migration Mechanism

HM-Keeper uses a high-performance page migration mechanism and migrates pages at the granularity of memory regions.

8.6.1 Performance Analysis of Page Migration Mechanism

Linux provides an API, *move_pages()*, for a privileged process to move a group of 4KB pages from a source memory node (or tier) to a target memory node (or tier). *move_pages()* consists of four main steps:

1. Allocate new memory pages in the target memory node;
2. Unmap pages to migrate (including invalidating PTE);

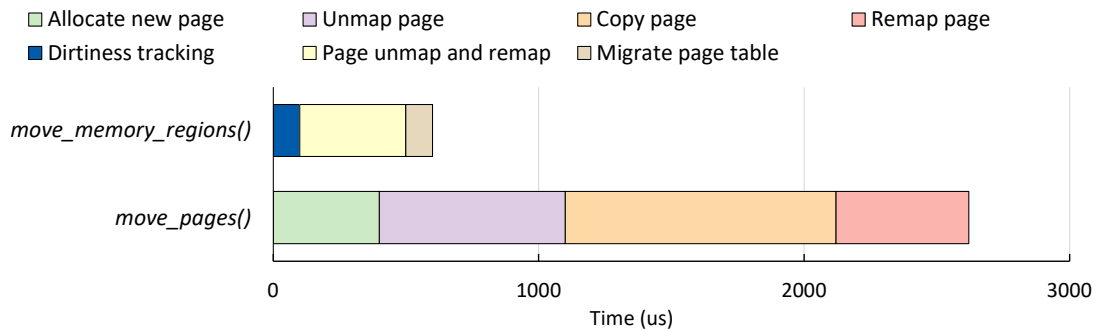


Figure 8.4: Performance breakdown for migration mechanisms.

3. Copy pages from source to target memory nodes;
4. Map new pages (including updating PTE).

Figure 8.4 shows the performance of migrating a 2MB memory region from the fastest memory tier to the slowest memory tier with *move_pages()* in the Optane-based platform. Copying pages is the most time-consuming step, taking 40% of total time. *move_pages()* moves 4K-sized pages sequentially, causing large page migration overhead. Although recent work [234] enables multi-threaded page copy to fully utilize memory bandwidth, copying pages is still the performance bottleneck, especially when moving a large memory region.

8.6.2 Adaptive Page Migration Schemes

Asynchronous page copy. We introduce an asynchronous page copy mechanism to reduce page copy overhead. In *move_pages()*, all of the four steps are performed one after another synchronously. But in the asynchronous page copy, the thread that triggers migration (named the *main thread* in the rest of the discussion) launches one or more helper threads to run the steps (1) and (3); the main thread runs the steps (2) and (4), and then waits for the helper thread(s) to join. With the asynchronous page copy, it is possible that copying a page happens before invalidating its PTE and the page is modified in the source memory node after copying the page. For such a case, copying the page must happen again to update the page in the target memory node. Hence, the asynchronous page copy has limitation: some pages have to be copied twice, which can be costly. We introduce an adaptive page-migration mechanism to address the above limitation.

Adaptive page migration. For read-intensive pages, the asynchronous page copy is likely to bring performance benefit. However, for write-intensive pages, due to repeated data copy, it is likely that the asynchronous page copy performs worse than the synchronous. Hence, HM-

Keeper chooses suitable migration mechanism based on the write intensity of pages. In particular, HM-Keeper uses the asynchronous page copy by default. But whenever any page in the memory region for migration is written after the asynchronous page copy starts, HM-Keeper switches to the synchronous page copy. To track page dirtiness, HM-Keeper utilizes PTE access bits and page faults, discussed in Section 8.7.

Other optimizations. (1) *Concurrent page copy and parallel page copy.* HM-Keeper uses multiple threads to copy pages, and bidirectional page copy between two memory tiers (i.e., from A to B and from B to A) in parallel. Similar optimization can be found in [234].

(2) *Migration of PTE.* Recent works [12, 149] reveal that the page table is distributed across all memory tiers, and the remote page-table walk can happen frequently in a large memory system, degrading performance. In HM-Keeper, a memory region's corresponding PTEs take at least one page, and HM-Keeper uses the synchronous page copy to migrate PTEs to the memory tier where the migrating memory region moves.

We implement the above optimizations and introduce a new API called *move_memory_regions()*. In this implementation, tracking page dirtiness, performing page map/unmap, and migrating PTEs are still on the critical path, but the most time-consuming page copying could be performed off the critical path. Figure 8.4 presents the performance of *move_memory_regions()* migrating 2MB memory region using the same setting as for *move_pages()*, and excludes the overhead of page copying (and page allocation in the step (1), using asynchronous page allocation). *move_memory_regions()* is 4.37x faster than *move_pages()* in this case. Section 8.8.5 shows more results.

8.7 Implementation

We implement the adaptive memory profiling as a kernel module that periodically scans the page table based on adaptive page sampling. This kernel module takes a process ID as input, performs memory profiling, and saves the profiling result in a shared memory space. Profiling results are stored in a table, where each record contains a memory region ID, hotness indication in the current profiling interval, and the EMA of hotness indication of prior profiling intervals. The region ID is generated based on the start address of the memory region.

We implement the holistic page management as a daemon service at the user space. The daemon service executes with the application and calls the kernel module for profiling at the beginning of each profiling interval. At the end of each profiling interval, the service reads the shared memory space to collect the profiling results. Specifically, with overhead control, the

profiling module ensures profiling finished in a profiling interval. The service then makes the migration decision and performs migration using *move_memory_regions()*.

move_memory_regions() takes the same input as Linux *move_pages()*, but implements the adaptive migration mechanism. It detects page dirtiness during the migration by setting a reserved bit in PTE, such that when any page in the memory region is written, a write protection fault is triggered. Leveraging a user-space page fault handler, *move_memory_regions()* tracks writes, and decides whether to stop the asynchronous page copy and switch to the synchronous mechanism.

HM-Keeper runs on Linux 5.4.0. HM-Keeper has 709 LOC, 1421 LOC, and 330 LOC, respectively, to implement the profiling kernel module, memory management daemon service, and the new migration call.

8.8 Evaluation

8.8.1 Experimental Setup

Testbed. We evaluate HM-Keeper on a two-socket machine based on Intel Optane DC persistent memory module (PMM). This machine has four memory tiers (see Section 8.2.1 for details). We also emulate another multi-tiered large memory system based on the Optane machine for evaluation. The emulated system has four tiers, and their latency and bandwidth are different from those in the original Optane machine. For the emulation, we launch two memhog instances in each memory tier to continuously inject memory access traffic. Memhog is an artificial memory-intensive workload in Linux used in prior studies to emulate heterogeneous memory systems [148, 234]. Table 8.1 summarizes the two evaluation platforms. Unless indicated otherwise, Optane uses App Direct Mode, which allows software-based page management, and we report results on the Optane-based machine (not the emulation platform). We use *madvise* for Transparent Hugepage Support (THP) [203], which uses 2MB as page size for contiguous memory allocation. This configuration is typical in large memory systems. We set the profiling overhead constraint to 5% and the profiling interval to 10 seconds. This setting is similar to existing works and production environments [14, 78, 151].

Workloads. Common large-memory workloads, including in-memory database, graph analysis, and data sorting (see Table 8.2) are used. The memory footprint of these workloads is larger than the first two memory tiers (fast memories), allowing us to effectively evaluate page management on all four tiers. Unless indicated otherwise, workloads were evaluated with eight application threads.

Baselines for performance comparison. Nine state-of-the-art application-agnostic memory management solutions are used for evaluation.

- *Hardware-managed memory caching (HMC)* uses the fast memories as hardware-managed cache for slow memories. We use Memory Mode in Optane.
- *First-touch NUMA* is a common NUMA allocation policy. It allocates pages in a memory tier close to the running task that first touches the pages. It does not migrate pages after memory allocation and does not track page hotness.
- *AutoNUMA* [41] is a solution established well in Linux.
- *Memory-optimizer* [84] is a solution from Intel.
- *Memory-tiering* [85] is a solution from Intel based on an extension to AutoNUMA.
- *AutoTiering* [98] is a state-of-the-art solution for multi-tiered memory systems based on AutoNUMA. To enable its best performance, we enable its opportunistic promotion and migration (OPM) and background demotion.
- *HeMem* [176] is a state-of-the-art solution for two-tiered PM-based HM. HeMem leverages hardware performance counters to find hot pages and promotes hot page to the fast memory tier.
- *Thermostat* [14] is a solution for two-tiered HM. It randomly samples pages for profiling. It allocates all pages in fast memory and selectively moves them to slow memory to save production cost, which cannot work for our use cases where the application footprint is larger than fast memory. We do not evaluate page migration of Thermostat but evaluate its profiling method.
- *Nimble* [234] is a highly optimized page migration mechanism using bi-direction page copy and parallel page copy. We use it to evaluate our page migration mechanism.

8.8.2 Overall Performance

Optane-based multi-tiered memory system. Figure 8.5 shows that HM-Keeper outperforms all the memory management solutions in the five workloads. We have six interesting observations.

(1) HM-Keeper outperforms HMC up to 52% (24% on average). HMC incurs write amplification when cache misses occur frequently [76], which causes unnecessary data movement and poor performance.

(2) HM-Keeper outperforms first-touch NUMA in all cases by up to 45% (20% on average). Without page migration, first-touch NUMA outperforms HMC on VoltDB and BFS, and outperforms AutoNUMA on Cassandra and BFS, indicating that page migration does not always bring

Table 8.1: Hardware overview of experimental system.

Evaluation Platform		
CPU	Intel Xeon Gold 6252 CPU x 2	
# of cores	24 x 2	
Last Level Cache	36608KB	
Fast Memory	96GB DDR4 DIMM x 2	
Slow Memory	756GB Optane DC PMM x 2	
Optane-based Multi-tiered Memory System		
Fast Mem Local Access (1st tier)	latency: 90ns	bw: 95 GB/s
Fast Mem Remote Access (2nd tier)	latency: 145ns	bw: 35 GB/s
Slow Mem Local Access (3rd tier)	latency: 275ns	bw: 35 GB/s
Slow Mem Remote Access (4th tier)	latency: 340ns	bw: 1 GB/s
Emulated Multi-tiered Memory System		
Fast Mem Local Access (1st tier)	latency: 198ns	bw: 95 GB/s
Fast Mem Remote Access (2nd tier)	latency: 315ns	bw: 35 GB/s
Slow Mem Local Access (3rd tier)	latency: 825ns	bw: 35 GB/s
Slow Mem Remote Access (4th tier)	latency: 1010ns	bw: 1 GB/s

Table 8.2: Workloads for evaluation.

Workloads	Descriptions	Mem	R/W
VoltDB [213]	A commercial in-memory database with TPC-C [109] using 5K warehouse.	300GB	1:1
Cassandra [22]	A highly-scalable partitioned row store with YCSB [40] (using update-heavy benchmark A).	400GB	1:1
BFS [160]	A parallel implementation of graph traversing and searching on a randomly generated graph with 0.9B nodes and 14B edges.	525GB	read-only
SSSP [160]	A high-performance parallel implementation of finding the shortest path between two vertices on a randomly generated graph with 0.9B nodes and 14B edges.	525GB	read-only
Spark [243]	A spark program running the TeraSort benchmark [75].	350GB	1:1

performance improvement. HMC performs worse because of unnecessary page movement discussed above. AutoNUMA has worse performance because it cannot effectively identify hot pages.

(3) HM-Keeper outperforms AutoNUMA by up to 72% (25% on average). AutoNUMA does not support page migration between memory tiers within the same socket (Memory-tiering, which is an extension to AutoNUMA, supports so), and hence fails to take full advantage of all memory tiers.

(4) HM-Keeper outperforms Memory-optimizer by up to 65% (32% on average). Lack of page migration across sockets, Memory-optimizer fails to fully leverage the 2nd fastest memory on a socket for the application on the other socket.

(5) HM-Keeper outperforms Memory-tiering by up to 45% (22% on average). Different from AutoNUMA and Memory-optimizer, Memory-tiering enables page migration across all memory tiers. However, it promotes hot pages tier by tier instead of using fast promotion as HM-Keeper,

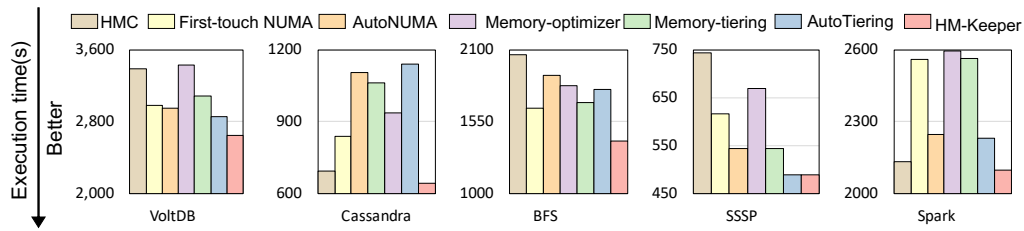


Figure 8.5: Performance comparison between existing solutions and HM-Keeper on the Optane-based multi-tiered memory system.

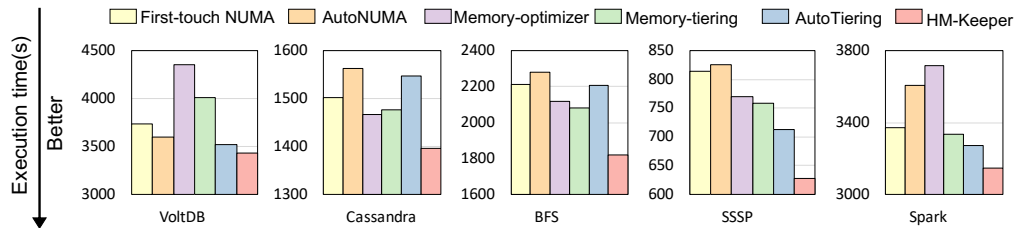


Figure 8.6: Performance comparison between existing solutions and HM-Keeper on the emulated multi-tiered memory system.

delaying the opportunity to improve performance using the fastest memory.

(6) HM-Keeper outperforms AutoTiering by up to 77% (20% on average). AutoTiering uses random sampling and opportunistic page demotion, which cannot effectively identify hot/cold pages for page migration.

Emulated multi-tiered memory system. Figure 8.6 shows the results. We do not evaluate HMC, because when HMC is used, fast memory tiers are hidden from software and we cannot inject latency into fast memory. On the emulated platform, HM-Keeper maintains the same performance trend as on Optane: HM-Keeper outperforms first-touch NUMA, AutoNUMA, Memory-optimizer, Memory-tiering and AutoTiering by 19%, 38%, 20%, 26% and 17% on average respectively.

Performance breakdown. We break down performance into application execution time, migration time, and profiling time, shown in Figure 8.7. The migration time is the migration overhead exposed to the critical path, excluding asynchronous page copying time. Figure 8.7 shows the results for Memory-tiering, AutoTiering and HM-Keeper, because they are the only solutions that can leverage all four memory tiers for migration. We add first-touch NUMA as performance baseline for comparison, because Memory-tiering, AutoTiering and HM-Keeper use it for memory allocation. In all cases, the profiling overhead falls within the profiling overhead constraint.

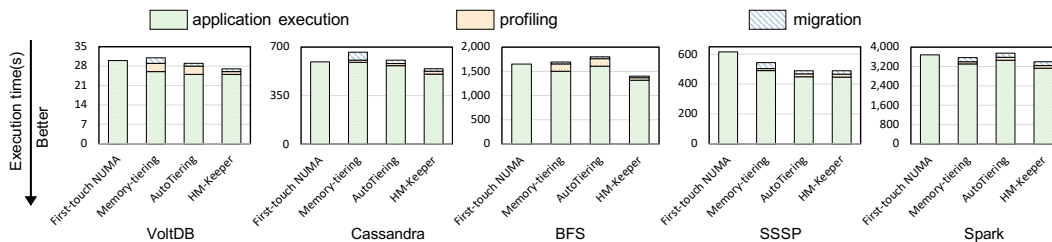


Figure 8.7: Breakdown of application execution time.

Table 8.3: The number of memory accesses when using voltDB.

# of memory accesses	mem-tiering	AutoTiering	HM-Keeper
tier 1	248M	258M	295M
tier 2	15M	34M	198K
tier 3	60M	30M	9M
tier 4	704K	2.5M	92K

With Memory-tiering and AutoTiering, the reduction of application execution time is less than or equal to the overhead brought by profiling and migration (see VoltDB and Cassandra). Hence, Memory-tiering and AutoTiering perform worse than first-touch NUMA without page migration.

Compared to Memory-tiering, HM-Keeper spends similar time in profiling but 3.5x less time in migration, reducing the total application execution time by 21% on average. Compared to AutoTiering, HM-Keeper again spends similar time in profiling but 1.25x less time in migration, and reduces the application execution time by 19% on average. Because of the effectiveness of page sampling and migration strategy, HM-Keeper obtains better performance.

Number of memory accesses. We count the number of memory accesses at each memory tier when running voltDB. We only report the results for Memory-tiering, AutoTiering, and HM-Keeper, because they are the only solutions that leverage all memory tiers for migration. Table 8.3 shows the results. We use Intel Processor Counter Monitor [86] to count the number of memory accesses and then exclude memory accesses caused by page migration. This counting method allows us to evaluate how many memory accesses from the application (not from page migration) happen on memories.

Table 8.3 shows that with HM-Keeper, the number of memory accesses happen in the fastest memory (top tier) is 20% and 14% more than with Memory-tiering and AutoTiering. This indicates that HM-Keeper effectively migrates frequently accessed pages to the fast memory for high performance.

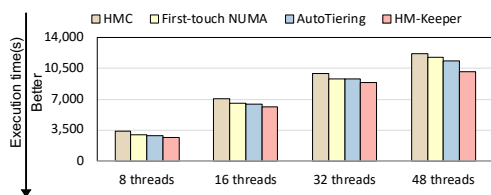


Figure 8.8: Execution time of VoltDB with different number of client threads.



Figure 8.9: Evaluating HM-Keeper with different large page sizes. We use the SSSP application for evaluation.

Scalability of HM-Keeper. We evaluate the scalability of HM-Keeper with VoltDB by increasing the number of application threads. Specifically, we increase the number of clients used in VoltDB. As the number of clients increases, the memory consumption increases from 300GB to 1TB. We compare the performance of HM-Keeper, HMC, first-touch NUMA, and AutoTiering. We evaluate AutoTiering since it has the second-best performance among all page management solutions we evaluate. Figure 8.8 shows that HM-Keeper consistently outperforms HMC, first-touch NUMA, and AutoTiering by 19%, 10%, and 8% on average as the number of application threads increases.

Evaluation with different page sizes. Figure 8.9 shows the results using 2MB and 1GB as the page size with THP enabled. We use SSSP for evaluation, because it has the largest memory consumption among all evaluated applications. The results confirm that HM-Keeper consistently outperforms other solutions even at different huge page sizes.

Evaluation on two-tiered HM and comparison with HeMem. We compare HM-Keeper with HeMem [176], a state-of-the-art solution for two-tiered HM. The evaluation is performed on a single socket with two memory tiers, using the benchmark GUPS [67] as in HeMem [176]. Figure 8.10 reports the results of running GUPS with 16 and 24 application threads, respectively. The results show that when the working set size fits in the fast memory tier (i.e., x values smaller than 1.0), HM-Keeper performs similarly to HeMem at 16 application threads and better at 24 application threads. Once the working set size exceeds the fast memory (i.e., x values larger than 1.0), HeMem fails to sustain application performance at 24 threads while HM-Keeper can still sustain higher performance at 24 threads than 16 threads. HM-Keeper performs better because its profiling method is able to quickly adapt to changes in memory accesses and identify more hot pages.

8.8.3 Effectiveness of Adaptive Profiling

We study profiling quality and overhead, and compare HM-Keeper with two sampling-based profiling methods (one is used in Memory-tiering, AutoNUMA, and AutoTiering, and the other is used in Thermostat). We use Memory-tiering and Thermostat for evaluation, and replace their migration strategy and mechanism with HM-Keeper's. This replacement ensures that we exclude the impact of migration on performance, and hence our comparison is fair.

The profiling method in Memory-tiering randomly chooses a 256MB virtual address space in each profiling interval, and then manipulates the present bit in each 4KB-page PTE in the chosen address space. This method tracks page accesses by counting page faults. The profiling method in Thermostat randomly chooses a 4KB page out of each 2MB memory region for profiling. This method manipulates page protection bits in PTE and leverages protection faults to count accesses.

Figure 8.11 shows the result. HM-Keeper outperforms Memory-tiering and Thermostat by 17% and 7% respectively. Thermostat has higher profiling overhead than Memory-tiering by 6x because the number of sampled pages for profiling in Thermostat is much larger than that in Memory-tiering. Thermostat has higher profiling overhead than HM-Keeper by 2.5x, since manipulating reserved bits in PTE and counting protection faults in Thermostat is more expensive than scanning PTE in Memory-tiering and HM-Keeper. Using Memory-tiering, the application execution time is longer than with HM-Keeper by 22%. This indicates that random sampling-based profiling is not as effective as our adaptive profiling method. The adaptive profiling, when choosing samples, considers both temporal and spatial locality, and aims to maximize profiling quality within a profiling overhead constraint, which is missing in random sampling.

We further evaluate three major profiling techniques in HM-Keeper (i.e., adaptive memory regions, adaptive page sampling, and profiling overhead control). We disable them one by one and then examine the performance difference. The last three groups of bars in Figure 8.11 show the evaluation results.

Evaluation of adaptive memory regions. The technique of adaptively merging and splitting memory regions aims to improve profiling quality. We disable it but respect the profiling overhead constraint. Figure 8.11 shows that the application execution time is 22% longer, although the profiling overhead constraint is met. Such a performance loss indicates that hot memory regions are not effectively identified without adaptive regions and hence placed in slow memory.

Evaluation of adaptive page sampling. This technique distributes samples between memory regions by using time-consecutive profiling results, which includes information on temporal locality. We disable this technique and randomly distribute samples between memory regions, and

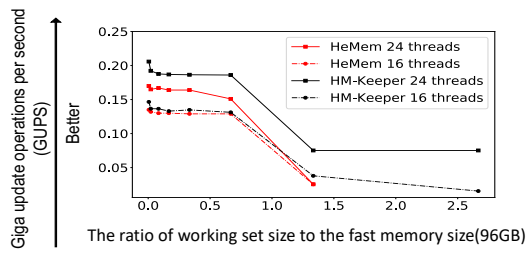


Figure 8.10: Evaluation of HM-Keeper on two-tiered HM and comparison with HeMem.

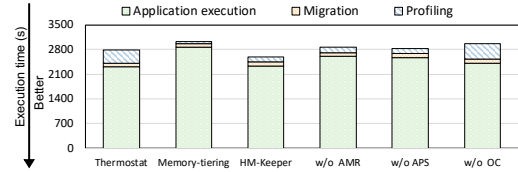


Figure 8.11: Evaluation of the effectiveness of adaptive memory regions (“AMR”), adaptive page sampling (“APS”), and profiling overhead control (“OC”) in the VoltDB execution time.

observe 21% performance loss in Figure 8.11. This indicates the importance of using temporal locality as the metric to guide the selection of samples.

Evaluation of profiling overhead control. We disable profiling overhead control by setting $\tau_1 = \tau_2 = 0$ (i.e., no merging/splitting memory regions) and removing the control of the number of page samples (num_ps in Equation 8.1). We observe that the profiling time is increased by 3x in Figure 8.11.

We further study the relation between profiling overhead and profiling quality. Figure 8.12 shows the result. We set profiling interval length as 5s and test a set of profiling overhead targets. As the profiling target increase from 1% to 10%, the application execution time decrease by 12%, which reveals that the profiling quality can be improved by increasing the number of samples in profiling. However, better profiling quality does leads to better system performance, since the performance benefit from more accurate profiling result may not able to cover extra profiling overhead we have to pay. As shown in figure 8.12, the end-to-end performance decreases by 7% as the profiling target increase from 5% to 10%.

8.8.4 Effectiveness of Migration Strategy

HM-Keeper uses a flat view on multi-tiered memory systems, and adopts the “fast promotion but slow demotion” strategy. To evaluate the effectiveness of this strategy, we change HM-Keeper to use a hierarchical view, where hot pages need multiple profiling intervals to reach the fastest memory. Figure 8.13 shows that the flat view performs 20% better than the hierarchical view because the fastest memory is used more effectively.

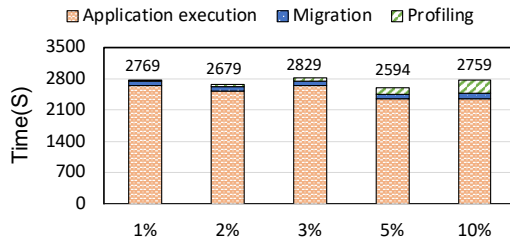


Figure 8.12: Execution time with setting various of profiling overhead targets when execute voltDB with HM-Keeper.

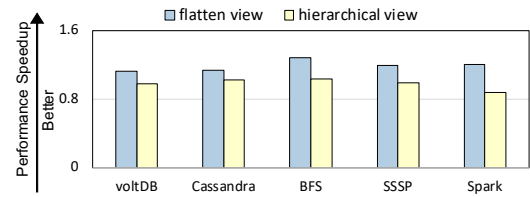


Figure 8.13: Comparison between flatten- and hierarchical view-based migration. Performance speedup is calculated based on the performance of first-touch NUMA.

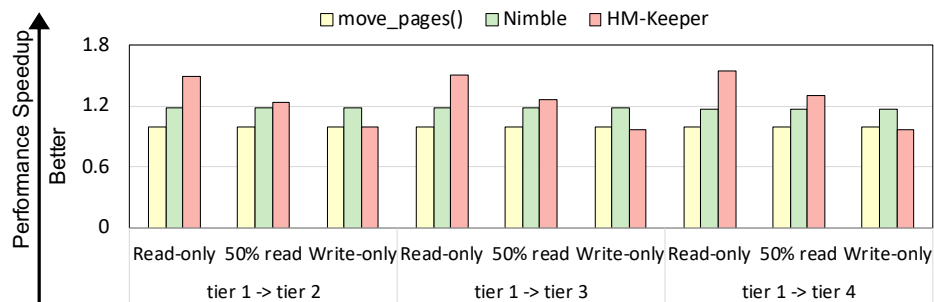


Figure 8.14: Performance comparison between Nimble, *move_pages()* and HM-Keeper in page migration mechanism. The performance speedup is calculated based on the performance of using *move_pages()*.

8.8.5 Effectiveness of Migration Mechanism

We use three microbenchmarks to evaluate the migration mechanisms in HM-Keeper, Nimble [234], and *move_pages()* in Linux. The microbenchmarks perform sequential read-only, 50% read (i.e., a sequential read followed by an update on an array element), and 100% sequential write on a 1GB array, respectively. The array is allocated and touched in a memory tier, and then migrated to another tier during the execution. Figure 8.14 shows the results. Migrating pages between the tiers 1 and 2, HM-Keeper's mechanism performs 40%, 23%, and -0.5% better than *move_pages()*, and performs 36% 4% and -6% better than Nimble, for read-only, 50% read, and write-only scenarios respectively. We see the same trend in other tiers. In general, for read-intensive pages, HM-Keeper's mechanism brings large performance benefit because of asynchronous page copy; for write-intensive pages, HM-Keeper's mechanism performs similar to *move_pages()* and Nimble, because of the overhead of tracking page dirtiness during the migration.

8.9 Conclusions

Emerging multi-tiered large memory systems bring new challenges to system software and applications. In this work, we study page management in multi-tiered large memory systems and pinpoint the fundamental limitations in existing solutions, i.e., non-scalable, low-quality memory profiling and unawareness of rich memory tiers. We present HM-Keeper, an application-transparent page management system customized for large memory systems. HM-Keeper is based on four design principles, i.e., scalable high-quality profiling, global view of all memory tiers, holistic migration decision and pattern-aware migration mechanisms. Our extensive evaluation of HM-Keeper against seven state-of-the-art solutions shows that HM-Keeper can largely outperform existing solutions on four-tiered large memory systems by 15%-78%.

Chapter 9

Conclusions and Future Directions

9.1 Conclusions

The emergence of big memory applications (e.g., AI/ML, large-scale scientific simulation, and in-memory database etc) and the emergence of new memory technologies (e.g., e.g., HBM, HMC, PCM, 3D-XPoint, ReRAM, fabric attached memory, and disaggregated memory) are calling the birth of big memory systems. We envision that the HM-based big memory systems become more and more common to see in parallel computing scenario.

The memory management in big memory systems is challenging, due to (1) the gap exists in inaccurate and unscalable memory access profiling methods in runtime systems and complex and irregular memory access patterns in applications; (2) making data movement decisions based on profiling results without violating fast memory capacity; and (3) inefficient data migration mechanism between different memory devices.

This dissertation tackles those challenges via a series of solutions **spanning different levels of the system stack**, leveraging memory management runtime techniques to improve the whole system throughput, which is equipped with big memory systems. Specifically,

- *HM-ANN* leverages a **co-design of algorithm and system** to map the hierarchical design of the graph-based approximate nearest neighbour(ANN) search algorithm to memory heterogeneity in Optane-based heterogeneous memory and enables billion-scale ANN search with 95% top-1 recall in less than one millisecond.
- *Sentinel* **coordinates OS and runtime-level profiling** to bridge the semantic gap between OS and applications, and enables accurate tensor-level profiling without ML model

modification. Powered by such tensor-level profiling method, *Sentinel* successfully avoids out-of-memory (OOM) issues for large static deep neuron network (DNN) models on GPU, and outperforms five state-of-the-art memory management solutions for DNN training.

- *ZeRO-Offload* breaks the trade-off between GPU memory saving and communication volume between GPU memory and CPU memory by **applying application domain knowledge in memory management runtime**, and enables 10X larger transformer-based model being trained on a single GPU and achieves near-linear speedup on up to 128 GPUs.
- *DyNN-Offload* uses a learned approach to resolve dynamism and predict access order of tensors in dynamic neural network (DyNN) training. **Learning from static program information** (i.e., DyNN model code snippets) **and dynamic runtime information** (DyNN inputs), *DyNN-Offload* enables fast inference while providing high prediction accuracy for tensor accesses. *DyNN-Offload* demonstrates the possibility of using a learned approach to remove dynamism and address complicated problems on performance optimization and analysis. *DyNN-Offload* outperforms state-of-the-art solutions by 2%-50% in terms of training time with the same GPU memory capacity and enables 8x larger model training without out of memory.
- *WarpX-PM* **applies algorithm knowledge** (i.e., particle in cell method) **in memory management runtime** and enables high resolution plasma simulation on Optane-based HM with high performance. *WarpX-PM* is implemented on a production-level mission-critical plasma simulation called Warp-X. *WarpX-PM* accelerates the execution of WarpX on HM by over 60% (compared with the case of no management).
- *HM-Keeper* **bridges the gap between memory management runtime and OS** with a new memory abstract called memory region, and further proposes a highly accurate, scalable memory profiling method, a global view guided memory migration strategy, and a high efficient memory region migration masochism on a multi-tiered big memory system. *HM-Keeper* largely outperforms existing page management solutions on large memory systems. Memory-consuming applications such as in-memory databases, billion-scale graph processing, and many other big data applications can greatly benefit from *HM-Keeper*.

9.2 Future Directions

Some potential future expeditions could be pursued to achieve the overarching goal as techniques proposed in this thesis of lowering the barriers for applications to use big memory systems easily and efficiently are as follows:

Learned memory management. Leveraging applications' domain knowledge can significantly simplify the identification of data access patterns, and achieve highly efficient memory management solutions for given applications. However, getting those domain knowledge is not free. Domain knowledge guided memory management makes sense for those frequently used applications (such as the nearest neighbor search) or mission-critical applications (such as some HPC applications). There are applications where domain knowledge cannot be easily obtained. In those applications, general-purpose (or application-agnostic) memory management is a better solution. The critical challenge of general-purpose memory management is to timely capture memory access patterns and make data placement decision accordingly. Therefore I proposal using machine learning methods to learn implicit knowledge of applications to guide memory profiling. The learning process should happens in the whole system stack, from architecture, OS, runtime, compiler to application and algorithm.

Memory management for fabric attached memory based big memory systems. fabric attached memory technologies such as CXL and Gen-Z are emerging, which enables I/O operations at byte (instead of block) granularity, and provides fine-grained, microsecond-level interconnect latency. Such new interconnect technologies bring challenges and opportunities to HM systems: the direct accesses will be enabled among host memory, remote memory, and accelerators/CXL devices memory without any memory page movement.

With the change of memory access methods to remote memory/devices, the mismatch between memory access granularity and data movement (between memory components in HM) granularity must be revised, and the existing applications and algorithms must be revisited. For example, the application demands fine-grained (cache block level), low latency, transparent data access for high performance. However, the existing data movement is often built upon the virtual memory system at the page level, which incurs unnecessary data movement due to page-level false sharing and large performance degradation due to expensive page fault handling.

In my future research, I plan to explore how to leverage the cache-coherent feature provided by fabric attached memory to reduce the communication happened between different memory devices. A large number of communication intensive applications, such as big data applications (e.g., Spark-based big data sorting and big in-memory database), can benefit from this research.

Accelerating computation using memoization on big memory systems. Big memory systems brings memory capacity of several TB per machine. I envision that the memory capacity accessible to a server will continue increasing, given the emergence of persistent memory and disaggregated memory techniques. Such a large memory capacity brings new opportunities to improve application performance.

In particular, the existing applications and algorithms are designed for a server with limited memory capacity - applications and algorithms often use recomputation to save memory consumption. We can use memoization to leverage the big memory systems to improve application performance. The memoization technique, in nature, stores results of expensive computation to a data structure, such as a lookup table, such that when the same or similar computation happens, the results can be returned without performing expensive computation. With a control of computation approximation, the memory can be used to memorize intermediate computation results and save computation time.

The goal of my research in this topic is to provide an application-transparent middleware to enable memoization on HM-based big memory platforms for some applications. The memoization technique has been studied in existing work. However, most of efforts cannot work well when applied to the big memory systems since their memoization data structure and data-access granularity do not consider latency, bandwidth, and capacity characteristics in big memory systems with extreme heterogeneity. There are a large number of computation-intensive scientific applications and algorithms in HPC (such as molecular dynamics simulation, quantum chemistry applications, and machine learning models) can benefit from this research.

Bibliography

- [1] Key/value datastore for persistent memory. <https://github.com/pmem/pmemkv>.
- [2] Redis, enhanced to use persistent memory - limited prototype. <https://github.com/pmem/redis/tree/3.2-nvml>.
- [3] The Stanford Question Answering Dataset (SQuAD) leaderboard. <https://rajpurkar.github.io/SQuAD-explorer/>.
- [4] FAISS. <https://github.com/facebookresearch/faiss/tree/master/benches>, 2017.
- [5] ResNet in TensorFlow. <https://github.com/wenxinxu/resnet-in-tensorflow>, 2017.
- [6] A TensorFlow Implementation of Deep Convolutional Generative Adversarial Networks. <https://github.com/carpedm20/DCGAN-tensorflow>, 2018.
- [7] Navigating Spreading-out Graph For Approximate Nearest Neighbor Search. <https://github.com/ZJULearning/nsg>, 2018.
- [8] Rocksdb: A persistent key-value store for fast storage environments. <https://rocksdb.org/>, 2019.
- [9] TensorFlow code and pre-trained models for BERT. <https://github.com/google-research/bert>, 2019.
- [10] TensorFlow models. <https://github.com/tensorflow/models>, 2019.
- [11] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz,

- Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.
- [12] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.
- [13] Neha Agarwal, David Nellans, Mark Stephenson, Mike O’Connor, and Stephen W. Keckler. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.
- [14] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi’an, China, April 8-12, 2017*, 2017.
- [15] Ibrahim Umit Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. A Machine Learning Framework to Improve Storage System Performance. In *ACM Workshop on Hot Topics in Storage and File Systems*, 2021.
- [16] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*, 2018.
- [17] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Crystal gazer: Profile-driven write-rationing garbage collection for hybrid memories. In *Abstracts of the 2019 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS ’19*, 2019.
- [18] Laurent Amsaleg and Herve Jegou. Datasets for approximate nearest neighbor search. <http://corpus-texmex.irisa.fr/>, 2010.

- [19] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Learning to Compose Neural Networks for Question Answering. In *North American Chapter of the Association for Computational Linguistics Conference (NAACL)*, 2016.
- [20] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. Sap hana adoption of non-volatile memory. *Proc. VLDB Endow.*, 10(12), August 2017.
- [21] Julian T. Angeles, Mark Hildebrand, Venkatesh Akella, and Jason Lowe-Power. Investigating Hardware Caches for Terabyte-scale NVDIMMs. In *Annual Non-Volatile Memories Workshop*, 2021.
- [22] Apache. Open Source NoSQL Database. <https://cassandra.apache.org/>, 2021.
- [23] Artem Babenko and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.
- [24] Artem Babenko and Victor S. Lempitsky. Improving Bilayer Product Quantization for Billion-Scale Approximate Nearest Neighbors in High Dimensions. *CoRR*, abs/1404.1831, 2014.
- [25] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In *International Conference on Learning Representations (ICLR)*, 2015.
- [26] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
- [27] Abanti Basak, Zheng Qu, Jilan Lin, Alaa R Alameldeen, Zeshan Chishti, Yufei Ding, and Yuan Xie. Improving streaming graph processing performance using input knowledge. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1036–1050, 2021.

- [28] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD 1990*, pages 322–331, 1990.
- [29] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, September 1975.
- [30] Charles K Birdsall and A Bruce Langdon. *Plasma physics via computer simulation*. CRC press, 2004.
- [31] Alan Bivens, Parijat Dube, Michele Franceschini, John Karidis, Luis Lastras, and Mickey Tsao. Architectural Design for Next Generation Heterogeneous Memory Systems. In *International Memory Workshop*, 2010.
- [32] Kevin J Bowers, BJ Albright, L Yin, B Bergen, and TJT Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5):055703, 2008.
- [33] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.
- [34] Yu Cao, Wei Bi, Meng Fang, and Dacheng Tao. Pretrained language models for dialogue generation with multiple input sources, 2020.
- [35] Laura Carrington, Mustafa M. Tikir, Catherine Olschanowsky, Michael Laurenzano, Joshua Peraza, Allan Snavely, and Stephen Poole. An Idiom-Finding Tool for Increasing Productivity of Accelerators. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2011.
- [36] Cheng Chen, Jun Yang, Mian Lu, Taize Wang, Zhao Zheng, Yuqiang Chen, Wenyuan Dai, Bingsheng He, Weng-Fai Wong, Guoan Wu, Yuping Zhao, and Andy Rudoff. Optimizing In-Memory Database Engine for AI-Powered Online Decision Augmentation Using Persistent Memory. In *Proceedings of the VLDB Endowment*, 2021.

- [37] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015.
- [38] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv: Learning*, 2016.
- [39] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. Atmem: Adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2020*, 2020.
- [40] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010.
- [41] Jonathan Corbet. AutoNUMA: the Other Approach to NUMA scheduling. <http://lwn.net/Articles/488709/>.
- [42] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc’auelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.
- [43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [44] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. Sage: Parallel semi-asymmetric graph algorithms for nvrams. *Proc. VLDB Endow.*, 13(9):1598–1613, May 2020.
- [45] Bang Di, Daokun Hu, Zhen Xie, Jianhua Sun, Hao Chen, Jinkui Ren, and Dong Li. TLB-pilot: Mitigating TLB Contention Attack on GPUs with Microarchitecture-Aware Scheduling. *Transactions on Architecture and Code Optimization*, 19(1), 2021.
- [46] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.*, 2013.

- [47] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Notices*, volume 50, pages 379–390. ACM, 2015.
- [48] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *International Symposium on High-Performance Parallel and Distributed Computing*, 2019.
- [49] Thaleia Dimitra Doudali, Daniel Zahka, and Ada Gavrilovska. Cori: Dancing to the right beat of periodic data movements over hybrid memory systems. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 350–359, 2021.
- [50] Matthijs Douze, Hervé Jégou, and Florent Perronnin. Polysemous codes. In *ECCV*. Springer, 2016.
- [51] Matthijs Douze, Alexandre Sablayrolles, and Hervé Jégou. Link and Code: Fast Indexing With Graphs and Compact Regression Codes. In *CVPR*, 2018.
- [52] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proc. 11th European Conf. Computer Systems (EuroSys '16)*, 2016.
- [53] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 15. ACM, 2016.
- [54] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. Return of the lernaean hydra: Experimental evaluation of data series approximate similarity search. *Proc. VLDB Endow.*, 13(3):403–420, 2019.
- [55] Jianwei Feng and Dong Huang. Optimal gradient checkpoint search for arbitrary computation graphs, 2021.
- [56] Pradeep Fernando, Ada Gavrilovska, Sudarsun Kannan, and Greg Eisenhauer. Nvstream: Accelerating hpc workflows with nvram-based transport for streaming objects. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '18*, pages 231–242, New York, NY, USA, 2018. ACM.

- [57] Samuel F. Martins, Ricardo A. Fonseca, Luís O. Silva, Wei Lu, and Warren B. Mori. Numerical Simulations of Laser Wakefield Accelerators in Optimal Lorentz Frames. *Computer Physics Communications*, 181(5):869–875, 2010.
- [58] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. In *VLDB'19*, 2019.
- [59] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946–2953, 2013.
- [60] Sanjay Ghemawat. Tcmalloc : Thread-caching malloc. <https://gperftools.github.io/gperftools/tcmalloc.html>. Accessed October 28, 2019.
- [61] Michael Giardino, Kshitij Doshi, and Bonnie Ferri. Soft2LM: Application Guided Heterogeneous Memory Management. In *International Conference on Networking, Architecture, and Storage (NAS)*, 2016.
- [62] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory, 2019.
- [63] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *VLDB'99*, pages 518–529, 1999.
- [64] Google. PyTorch. <https://pytorch.org/>.
- [65] Google. Tensorflow Bucketing. <https://www.tensorflow.org/versions/r0.12/>, 2017.
- [66] David P Grote, Alex Friedman, Jean-Luc Vay, and Irving Haber. The warp code: modeling high intensity ion beams. In *AIP Conference Proceedings*, volume 749, pages 55–58. American Institute of Physics, 2005.
- [67] GUPS. Giga Updates Per Second. <http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/>, 2021.
- [68] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast Approximate Nearest-Neighbor Search with k-Nearest Neighbor Graph. In *IJCAI 2011*, pages 1312–1317, 2011.

- [69] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, and Woongki Baek. Hotness- and Lifetime-Aware Data Placement and Migration for High-Performance Deep Learning on Heterogeneous Memory Systems. *IEEE Transactions on Computers*, 69(3).
- [70] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic neural networks: A survey. *CoRR*, abs/2102.04906, 2021.
- [71] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [72] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training, 2018.
- [73] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning Memory Access Patterns. In *International Conference on Machine Learning*, 2018.
- [74] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.
- [75] Ewan Higgs. Spark-terasort. <https://github.com/ehiggs/spark-terasort>, 2018.
- [76] Mark Hildebrand, Julian T. Angeles, Jason Lowe-Power, and Venkatesh Akella. A Case Against Hardware Managed DRAM Caches for NVRAM Based Systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021.
- [77] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, 2020.

- [78] Takahiro Hirofuchi and Ryousei Takano. Raminator: Hypervisor-based virtualization for hybrid main memory systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, 2016.
- [79] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.
- [80] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. Multi-Scale Dense Networks for Resource Efficient Image Classification. In *International Conference on Learning Representations (ICLR)*, 2018.
- [81] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2018.
- [82] Amazon Inc. Amazon EC2 High Memory Instances with 6, 9, and 12 TB of Memory, Perfect for SAP HANA. <https://aws.amazon.com/blogs/aws/now-available-amazon-ec2-high-memory-instances-with-6-9-and-12-tb-of-memory-perfectfor-sap-hana/>, September 2018.
- [83] Intel. Big Memory Breakthrough for Your Biggest Data Challenges. <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html>.
- [84] Intel. Intel Memory Optimizer. <https://github.com/intel/memory-optimizer>, 2019.
- [85] Intel. Intel Memory Tiering. <https://lwn.net/Articles/802544/>, 2021.
- [86] Intel. Intel Processor Counter Monitor. <https://github.com/opcm/pcm>, 2021.
- [87] 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. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

- [88] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems (MLSys)*, 2020.
- [89] Herve Jegou, Matthijs Douze, and Cordelia Schmid. In *Product Quantization for Nearest Neighbor Search*, 2011.
- [90] Herve Jegou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP 2011*, 2011.
- [91] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *SysML Conferenc*, 2019.
- [92] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Trans. Archit. Code Optim.*, 15(3), September 2018.
- [93] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *CoRR*, abs/1702.08734, 2017.
- [94] Yannis Kalantidis and Yannis S. Avrithis. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR 2014*, pages 2329–2336, 2014.
- [95] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *International Symposium on Computer Architecture*, 2017.
- [96] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.
- [97] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.
- [98] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *USENIX Annual Technical Conference*, 2021.

- [99] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.
- [100] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization, 2021.
- [101] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, 2020.
- [102] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation. In *International Conference on Neural Information Processing Systems*, 2019.
- [103] Renaud Lachaize, Baptiste Lepers, and Vivien Quema. MemProf: A memory Profiler for NUMA multicore systems. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 53–64, Boston, MA, June 2012. USENIX Association.
- [104] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.
- [105] Lawrence Berkeley National Lab. Laser-Plasma Accelerators for High-Energy Physics. <https://bella.lbl.gov/research/bella-center-research-high-energy-physics/>.
- [106] Tung D Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. TFLMS: Large model support in tensorflow by graph rewriting. In *arXiv preprint arXiv:1807.02037*, 2018.
- [107] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, 2019.
- [108] Victor Lempitsky. The inverted multi-index. In *CVPR '12*, pages 3069–3076, 2012.

- [109] Scott T. Leutenegger and Daniel Dias. A Modeling Study of the TPC-C Benchmark. In *SIGMOD Record*, 1993.
- [110] Daixuan Li and Jian Huang. A learning-based approach towards automated tuning of ssd configurations, 2021.
- [111] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, 2020.
- [112] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Wenjie Zhang, and Xuemin Lin. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering*, 2019.
- [113] Liang Liang, Rong Chen, Haibo Chen, Yubin Xia, KwanJong Park, Binyu Zang, and Haibing Guan. SLAQ: Quality-Driven Scheduling for Distributed Machine Learning. In *ACM Symposium on Cloud Computing (SoCC)*, 2017.
- [114] Xiaodan Liang, Xiaohui Shen, Jiashi Feng, Liang Lin, and Shuicheng Yan. Semantic object parsing with graph LSTM. *CoRR*, abs/1603.07063, 2016.
- [115] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. Elastic machine learning algorithms in amazon sagemaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 731–737, New York, NY, USA, 2020. Association for Computing Machinery.
- [116] Felix Xiaozhu Lin and Xu Liu. memif: Towards Programming Heterogeneous Memory Asynchronously. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
- [117] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime Neural Pruning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.
- [118] Linux. numactl-linux man page.

- [119] Linux. Automatic NUMA Balancing. <https://www.linux-kvm.org/images/7/75/01x07b-NumaAutobalancing.pdf>, 2014.
- [120] Jiawen Liu, Dong Li, Gokcen Kestor, and Jeffrey Vetter. Runtime concurrency control and operation scheduling for high performance neural network training. *International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.
- [121] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory. In *Principles and Practice of Parallel Programming*, 2021.
- [122] Jiawen Liu, Hengyu Zhao, Matheus A. Ogleari, Dong Li, and Jishen Zhao. Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach. In *IEEE/ACM International Symposium on Microarchitecture*, 2018.
- [123] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. Hierarchical hybrid memory management in os for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2223–2236, 2019.
- [124] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep Learning with Dynamic Computation Graphs. In *International Conference on Learning Representations (ICLR)*, 2017.
- [125] Jiaolin Luo, Luanzheng Guo, Jie Ren, Kai Wu, and Dong Li. Enabling Faster NGS Analysis on Optane-based Heterogeneous Memory. In *Proceedings of Supercomputing '20 (Posters)*, November 2020.
- [126] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-Based Memory Allocation for C++ Server Workloads. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [127] Alberto Magni, Dominik Grewe, and Nick Johnson. Input-aware auto-tuning for directive-based GPU programming. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 66–75, 2013.
- [128] Yury A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, 2020.

- [129] Mark Mansi and Michael Swift. Osim: Preparing System Software for a World with Terabyte-scale Memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 202.
- [130] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. *HotStorage'17*, page 4, USA, 2017. USENIX Association.
- [131] Jon Martindale. RAM has never been cheaper, but are the historic prices here to stay? . <https://www.digitaltrends.com/computing/why-is-ram-so-cheap/>, 2019.
- [132] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857. USENIX Association, July 2020.
- [133] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by GPU memory optimization on TensorFlow. In *ML Systems Workshop in NIPS*, 2017.
- [134] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks, 2017.
- [135] Gaurav Mitra, Beau Johnston, Alistair Rendell, Eric McCreath, and Jun Zhou. Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. pages 1107–1116, 05 2013.
- [136] Marius Muja and David G. Lowe. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *TPAMI 2014*, 36(11):2227–2240, 2014.
- [137] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-Efficient Pipeline-Parallel DNN Training. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2021.
- [138] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm, 2021.
- [139] Graham Neubig, Yoav Goldberg, and Chris Dyer. On-the-fly Operation Batching in Dynamic Computation Graphs. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

- [140] Bao Nguyen, Hua Tan, and Xuechen Zhang. Large-scale adaptive mesh simulations through non-volatile byte-addressable memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, 2017.
- [141] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. Analyzing memory accesses with modern processors. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery.
- [142] Mohammad Norouzi and David J. Fleet. Cartesian K-Means. In *CVPR 2013*, 2013.
- [143] Mohammad Norouzi, David J. Fleet, and Ruslan Salakhutdinov. Hamming distance metric learning. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1070–1078, 2012.
- [144] Nvidia. Automatic Mixed Precision for Deep Learning. <https://developer.nvidia.com/automatic-mixed-precision>, 2019.
- [145] Nvidia. Nouveau: Accelerated Open Source driver for NVidia cards. <https://nouveau.freedesktop.org/wiki/>, 2019.
- [146] Nvidia. Unified Memory. <https://devblogs.nvidia.com/unified-memory-in-cuda-6/>, 2019.
- [147] ONNX. ONNX Runtime. <https://onnxruntime.ai/>.
- [148] Mark Oskin and Gabriel H. Loh. A software-managed approach to die-stacked dram. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 188–200, 2015.
- [149] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. Fast Local Page-Tables for Virtualized NUMA Servers with vMitosis. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.
- [150] Raghavendra Pappagari, Piotr Żelasko, Jesús Villalba, Yishay Carmiel, and Najim Dehak. Hierarchical transformers for long document classification, 2019.

- [151] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*, Middleware '19, page 1–7, New York, NY, USA, 2019. Association for Computing Machinery.
- [152] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, 2019.
- [153] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic Locality and Context-based Prefetching Using Reinforcement Learning. In *International Symposium on Computer Architecture*, 2015.
- [154] I. B. Peng and J. S. Vetter. Siena: Exploring the design space of heterogeneous memory systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 427–440, Nov 2018.
- [155] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the intel optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*. ACM, 2019.
- [156] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. Rthms: A tool for data placement on hybrid memory system. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, 2017.
- [157] Ivy Bo Peng, Kai Wu, Jie Ren, Dong Li, and Maya Gokhale. Demystifying the performance of HPC scientific applications on nvm-based memory systems. In *IPDPS*, 2020.
- [158] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 891–905, New York, NY, USA, 2020. Association for Computing Machinery.
- [159] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an

- Efficient Dynamic Resource Scheduler for Deep learning Clusters. In *EuroSys Conference*, 2018.
- [160] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–14, 2018.
- [161] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.
- [162] S. Perarnau, J. A. Zounmevo, B. Gerofti, K. Iskra, and P. Beckman. Exploring Data Migration for Future Deep-Memory Many-Core Systems. In *IEEE International Conference on Cluster Computing (CLUSTER)*, 2016.
- [163] picsar. The picsar project.
- [164] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. Training large neural networks with constant memory using a new execution algorithm. June 2020.
- [165] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In F. Dehne, J. R. Sack, and N. Santoro, editors, *Algorithms and Data Structures*, pages 437–449, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.
- [166] PyTorch. PyTorch Profiler. <https://pytorch.org/blog/introducing-pytorch-profiler-the-new-and-improved-performance-tool/>, 2021.
- [167] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
- [168] Moinuddin K. Qureshi, Michele Franchescini, Vijayalakshmi Srinivasan, Luis Lastras, Bulent Abali, and John Karidis. Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling. In *MICRO*, 2009.

- [169] Moinuddin K. Qureshi, Viji Srinivasan, and Jude A. Rivers. Scalable High-Performance Main Memory System Using Phase-Change Memory Technology. In *ISCA*, 2009.
- [170] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- [171] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020.
- [172] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.
- [173] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. *CoRR*, abs/2104.07857, 2021.
- [174] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. In Jian Su, Xavier Carreras, and Kevin Duh, editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 2383–2392. The Association for Computational Linguistics, 2016.
- [175] Luiz Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *International Conference on Supercomputing*, 2011.
- [176] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.
- [177] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. Optimizing Large-Scale Plasma Simulations on Persistent Memory-based Heterogeneous Memory with Effective Data Placement Across Memory Hierarchy. In *International Conference on Supercomputing (ICS)*, 2021.
- [178] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

- [179] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *USENIX Annual Technical Conference*, 2021.
- [180] Jie Ren, Kai Wu, and Dong Li. Exploring Non-Volatility of Non-Volatile Memory for High Performance Computing Under Failures. In *IEEE International Conference on Cluster Computing*, 2020.
- [181] Jie Ren, Minjia Zhang, and Dong Li. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Neurips*, 2020.
- [182] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, 2016.
- [183] Gina Roos. Dram Prices Continue to Climb. <https://epsnews.com/2017/08/18/dram-prices-continue-climb/>, 2017.
- [184] Corby Rosset. Turing-nlg: A 17-billion-parameter language model by microsoft, 2020.
- [185] Deboleena Roy, Priyadarshini Panda, and Kaushik Roy. Tree-cnn: A hierarchical deep convolutional neural network for incremental learning, 2019.
- [186] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, November 2020.
- [187] Andy Rudoff. Programming Models for Emerging Non-Volatile Memory Technologies. 38(3), 2013.
- [188] Mehrzad Samadi, Amir Hormati, Mojtaba Mehrara, Janghaeng Lee, and Scott Mahlke. Adaptive input-aware compilation for graphics engines. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22, 2012.
- [189] SeongJae Park. DAMON: Data Access Monitor. <https://sjp38.github.io/post/damon/>.

- [190] Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the Effects of Data Parallelism on Neural Network Training. *Journal of Machine Learning Research*, 20, 2019.
- [191] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiyang Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *USENIX Symposium on Operating Systems Design and Implementation*, 2018.
- [192] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10435–10444, 2018.
- [193] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *International Conference on Learning Representations (ICLR)*, 2017.
- [194] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A Hierarchical Neural Model of Data Prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.
- [195] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.
- [196] Jeffrey Mark Siskind and Barak A. Pearlmutter. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, 33(4-6):1288–1330, Sep 2018.
- [197] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. Astra: Exploiting Predictability to Optimize Deep Learning. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

- [198] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Rand-nsg: Fast accurate billion-point nearest neighbor search on a single node. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 13748–13758, 2019.
- [199] Ilya Sutskever, Oriol Vinyals, and Quocviet Le. Sequence to Sequence Learning with Neural Networks. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2014.
- [200] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks, 2015.
- [201] Billy Tallis. The Intel Optane Memory(SSD) Preview: 32GB of Kaby Lake Caching. <http://www.anandtech.com/show/11210/the-intel-optane-memory-ssd-review-32gb-of-kaby-lake-caching>.
- [202] Roman Tezikov. PyTorch implementation of L2L execution algorithm. <https://github.com/TezRomach/layer-to-layer-pytorch>, 2020.
- [203] The Linux Kernel User’s and Administrator’s Guide. Transparent Hugepage Support. <https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html>.
- [204] Thomas Willhalm, Roman Dementiev, Patrick Fay. Intel performance counter monitor - a better way to measure cpu utilization, 2017.
- [205] Philippe Tillet and David Cox. Input-Aware Auto-tuning of Compute-bound HPC kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.
- [206] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *USENIX Annual Technical Conference*, 2020.
- [207] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30*:

Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, pages 5998–6008, 2017.

- [208] J-L Vay, A Almgren, J Bell, L Ge, DP Grote, M Hogan, O Kononenko, R Lehe, A Myers, C Ng, et al. Warp-x: A new exascale computing platform for beam–plasma simulations. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 909:476–479, 2018.
- [209] G. Velkoski, M. Gusev, and S. Ristov. The performance impact analysis of loop unrolling. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 307–312, 2014.
- [210] John P Verboncoeur. Particle simulation of plasmas: review and advances. *Plasma Physics and Controlled Fusion*, 47(5A):A231, 2005.
- [211] Oreste Villa, Mark Stephenson, David Nellans, and Stephen Keckler. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *IEEE/ACM International Symposium on Microarchitecture*, 2019.
- [212] Oreste Villa, Mark Stephenson, David Nellans, and Stephen Keckler. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In *International Symposium on Computer Architecture*, 2020.
- [213] voltDB. voltDB. <https://www.voltdb.com/>, 2021.
- [214] Daniel Waddington, Mark Kunitomi, Clem Dickey, Samyukta Rao, Amir Abboud, and Jantz Tran. Evaluation of intel 3d-xpoint nvdimm technology for memory-intensive genomic workloads. In *Proceedings of the International Symposium on Memory Systems, MEMSYS '19*, 2019.
- [215] Bin Wang, Bo Wu, Dong Li, Xipeng Shen, Weikuan Yu, Yizheng Jiao, and Jeffrey S. Vetter. Exploring Hybrid Memory for GPU Energy Efficiency through Software-Hardware Co-Design. In *PACT*, 2013.
- [216] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, 2019.

- [217] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, November 2020.
- [218] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '18, page 41–53, New York, NY, USA, 2018. Association for Computing Machinery.
- [219] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '18, 2018.
- [220] WX Wang, ZTWM Lin, WM Tang, WW Lee, S Ethier, JLV Lewandowski, G Rewoldt, TS Hahm, and J Manickam. Gyro-kinetic simulation of global turbulent transport properties in tokamak experiments. *Physics of Plasmas*, 13(9):092505, 2006.
- [221] Duncan J. Watts. *Small Worlds: The Dynamics of Networks Between Order and Randomness*. Princeton University Press, 1999.
- [222] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, 2019.
- [223] Shasha Wen, Lucy Cherkasova, Felix Xiaozhu Lin, and Xu Liu. ProfDP: A Lightweight Profiler to Guide Data Placement in Heterogeneous Memory Systems. In *International Conference on Supercomputing*, 2018.
- [224] Wikipedia. Jaccard Index.
- [225] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and Exploiting

- Flexible Task Assignment on GPU through SM-Centric Program Transformations. In *International Conference on Supercomputing (ICS)*, 2015.
- [226] Kai Wu, Yingchao Huang, and Dong Li. Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.
- [227] Kai Wu, Jie Ren, and Dong Li. Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Memory for Task Parallel Programs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018.
- [228] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18*, 2018.
- [229] Zhaofeng Wu, Ding Zhao, Qiao Liang, Jiahui Yu, Anmol Gulati, and Ruoming Pang. Dynamic sparsity neural networks for automatic speech recognition. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021.
- [230] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [231] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. MD-HM: Memoization-based Molecular Dynamics Simulations on Big Memory System. In *International Conference on Supercomputing (ICS)*, 2021.
- [232] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. IA-SpGEMM: An Input-Aware Auto-tuning Framework for Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2019.
- [233] Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P. Xing. Cavs: An Efficient Runtime System for Dynamic Neural Networks. In *Proceedings of USENIX Conference on USENIX Annual Technical conference (ATC)*, 2018.
- [234] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *ASPLOS*, 2019.

- [235] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.
- [236] Jie Amy Yang, Jianyu Huang, Jongsoo Park, Ping Tak Peter Tang, and Andrew Tulloch. Mixed-Precision Embedding Using a Cache. In *Proceedings of Machine Learning and Systems*, 2020.
- [237] Peter N. Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *SODA '93*, pages 311–321, 1993.
- [238] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *ICCD*, 2012.
- [239] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training. In *USENIX Annual Technical Conference (ATC 21)*, 2021.
- [240] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable Neural Networks. In *International Conference on Learning Representations (ICLR)*, 2019.
- [241] Seongdae Yu, Seongbeom Park, and Woongki Baek. Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems. In *International Conference on Supercomputing (ICS)*, 2017.
- [242] Zhibin Yu, Zhendong Bei, and Xuehai Qian. Datasize-aware High Dimensional Configurations Auto-tuning of In-Memory Cluster Computing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.
- [243] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.
- [244] Junzhe Zhang, Sai-Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient memory management for gpu-based deep learning systems. *CoRR*, abs/1903.06631, 2019.

- [245] Minjia Zhang and Yuxiong He. Grip: Multi-store capacity-optimized high-performance nearest neighbor search for vector search engine. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM '19*, page 1673–1682, New York, NY, USA, 2019. Association for Computing Machinery.
- [246] Wangyuan Zhang and Tao Li. Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.
- [247] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. Amrex: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37), 2019.
- [248] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XI*, page 177–188, New York, NY, USA, 2004. Association for Computing Machinery.
- [249] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.