AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems using Integer Linear Programming

Mark Hildebrand
University of California, Davis
mhildebrand@ucdavis.edu

Jawad Khan
Intel Corporation
jawad.b.khan@intel.com

Sanjeev Trika
Intel Corporation
sanjeev.trika@intel.com

Jason Lowe-Power
University of California, Davis
jlowepower@ucdavis.edu

Venkatesh Akella
University of California, Davis
akella@ucdavis.edu

Abstract

Memory capacity is a key bottleneck for training large scale neural networks. Intel® Optane™ DC PMM (persistent memory modules) which are available as NVDIMMs are a disruptive technology that promises significantly higher read bandwidth than traditional SSDs at a lower cost per bit than traditional DRAM. In this work we show how to take advantage of this new memory technology to minimize the amount of DRAM required without compromising performance significantly. Specifically, we take advantage of the static nature of the underlying computational graphs in deep neural network applications to develop a profile guided optimization based on Integer Linear Programming (ILP) called AutoTM to optimally assign and move live tensors to either DRAM or NVDIMMs. Our approach can replace 50% to 80% of a system's DRAM with PMM while only losing a geometric mean 27.7% performance. This is a significant improvement over first-touch NUMA, which loses 71.9% of performance. The proposed ILP based synchronous scheduling technique also provides 2x performance over using DRAM as a hardware-controlled cache for very large networks.

CCS Concepts • **Hardware → Analysis and design of emerging devices and systems**; *Memory and dense storage*; • **Computing methodologies** → *Machine learning*.

ACM Reference Format:
Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. 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), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378465

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378465

1 Introduction

Deep Neural Networks (DNNs) have been dramatically successful over the past decade across many domains including computer vision [29], machine translation and language modeling [40], recommendation systems [30], speech [46] and image synthesis [47], and real-time strategy game control [43]. This success has in turn led practitioners to pursue larger, more expressive models. Today, state of the art models in language modeling and translation have 100s of billions of parameters [38] which requires 100s of GB of active working memory for training. For instance, large models such as BigGAN [6] found significant benefits from increasing both model size and training batch size, and Facebook's recent DLRM recommendation system [30] contains orders of magnitude more parameters than conventional networks. Additionally, to reach beyond human-level accuracy these models are expected to grow even larger with possibly 100× more parameters [22]. The large memory footprints of these models limits training to systems with large amounts of DRAM which incur high costs.

As the memory capacity demands of DNN training are growing, new high density memory devices are finally being produced. Specifically, Intel® Optane™ DC Persistent Memory Modules (PMM) [15, 25] can now be purchased and are up to 2.1× lower price per capacity than DRAM. These devices are on the main memory bus, allowing applications direct access via load and store instructions and can be used as working memory. Thus, in this paper we ask the question "*what are the design tradeoffs of using PMM in training large DNN models, and more specifically, can PMM be used as a DRAM replacement when training for large DNN models?*"

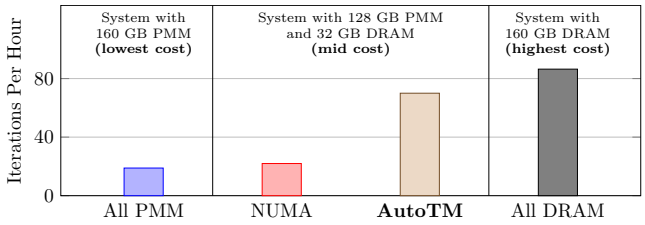


Figure 1. Performance of Inception v4. Batch size of 1472.

Figure 1 shows the training performance for three different memory systems: an all PMM system (lowest cost), an all DRAM system (highest cost), and a heterogeneous system (moderate cost). The all PMM bar shows that naively replacing DRAM with PMM results in poor performance (about 5× slowdown) for training large DNN models. The first-touch NUMA [27] bar shows that current system support for heterogeneous memory is lacking, providing only a small benefit over the all PMM case. However, AutoTM provides 3.7× speedup over the PMM case and is within 20% of the all DRAM system. Thus, we find that a small fraction of DRAM reduces the performance gap between PMM and DRAM, but only if we use *smart data movement*.

Use of heterogeneous memory to reduce DRAM has been studied in the past. Facebook has used SSDs to reduce the DRAM footprint of databases [13]. Bandana [14] uses SSD based persistent memory to store deep learning embedding tables [10] with DRAM as a small software cache. In the context of machine learning, vDNN [36], moDNN [8], and SuperNeurons [44] develope system-specific heuristics to tackle heterogeneous memory management between the GPU and CPU to overcome the low memory capacity of GPUs. Furthermore, future HPC systems will be increasingly heterogeneous with DRAM, PMM, and HBM [35], so we need a solution that is general and automatic.

In this paper we introduce AutoTM—a framework to automatically move DNN training data (tensors) between heterogeneous memory devices. AutoTM enables training models with 100s of billions of parameters and/or with large batch sizes efficiently on a single machine. We exploit the static nature of DNN training computation graphs to develop an Integer Linear Programming (ILP) [37] formulation which takes a profile driven approach to automatically optimize the location and movement of intermediate tensors between DRAM and PMM given a DRAM capacity constraint.

We evaluate the effectiveness of AutoTM on a real system with Optane PMM by implementing our approach in the nGraph compiler [11]. Our experiments show that naive use of PMM is not effective, but intelligent use of PMM and DRAM is required. Furthermore, using initial public pricing information, we evaluate the cost-performance benefits DRAM-PMM based systems. We show that ratios of $8:1$ or $4:1$ of PMM to DRAM can be more cost effective than only DRAM or only PMM.

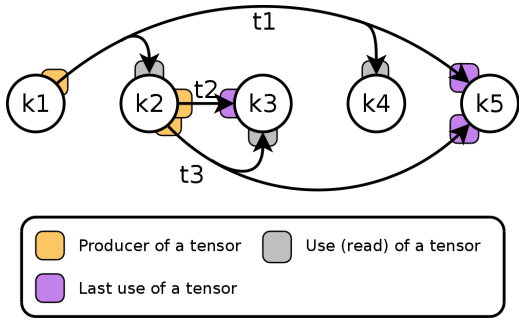


Figure 2. A simple example of a computation graph. The k nodes are the compute kernels in the graph and t edges (tensors) show the data dependency between kernels. Intermediate tensors have a finite live range that can be exploited to reduce the memory footprint of the computation graph.

We also compare our approach to the existing hardware DRAM cache implemented in current Intel platforms [25] and find AutoTM offers up to 2× performance improvement over hardware-managed caching.

Finally, we demonstrate that AutoTM can be further generalized beyond PMM-DRAM heterogeneity by applying AutoTM to CPU-GPU systems. The approach taken by AutoTM uses minimal problem specific heuristics and is thus a general approach toward memory management for many different heterogeneous systems.

The paper is organized as follows. In Section 2 we present a quick overview of training deep neural networks and Intel's Optane DC PMM. In Section 3 we will present the details of AutoTM and in Section 4 we will describe implementation details, followed by our evaluation methodology in Section 5, and the main results in Section 6. We will present extensions to AutoTM in Section 7 and conclude with related work and directions for future work.

2 Background

2.1 Deep Learning Training

Deep neural networks (DNNs) are often trained using a backward propagation algorithm [28] and an optimizer such as stochastic gradient descent. Popular deep learning frameworks such as Tensorflow [1] and nGraph [11] implement DNNs as a computation graph where each vertex or node in the computation graph represent some computational **kernel**. Common kernels include convolutions (CONV), pooling (POOL), matrix multiplication, and recurrent cells such as LSTM or GRU. Each kernel has its own characteristics such as number of inputs, number of outputs, computation time, and computational complexity. Directed edges in the computation graph between kernels denote data or control dependencies between kernels. An edge representing a data dependency is associated with a **tensor**, which we consider to be a contiguous region of memory with a known size.

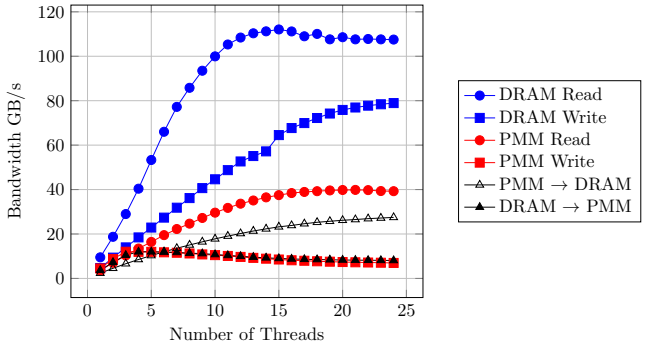


Figure 3. Read and write bandwidths between DRAM and PMM. All operations were performed using AVX512 load and stores. Copies between DRAM and PMM were done using streaming load and store intrinsics.

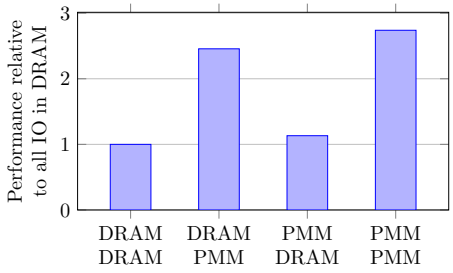


Figure 4. Execution time of a CONV kernel with input (upper label) and output (lower label) feature maps varied between DRAM and PMM. The performance of the kernel is largely unaffected by the location of the output feature map. The CONV kernel had a filter size (3, 3, 128, 128) and a input feature map size of (112, 112, 128, 128) and was executed using 24 threads.

Figure 2 shows a simple example computation graph with 5 kernels and 3 tensors. Nodes in the graph are compute kernels, each with zero or more inputs and outputs. The inputs and outputs of a kernel are immutable tensors. Each tensor is annotated with its producing kernel, each user of the tensor, and the last user of the tensor. After its last use, a tensor's memory may be freed for future tensors.

We focus on the case where the computation graph describing the training iteration is static. That is, the computation graph contains no data-dependent control behavior and the sizes of all intermediate data is known statically at compile time. While many DNN graphs can be expressed statically, there are some networks that exhibit data-dependent behavior [38]. We leave extending AutoTM to dynamic computation graphs as future work.

2.2 Intel Optane DC PMM

3D XPoint is a resistive memory technology developed by Intel [19] that was initially introduced in the form a SSD called Optane SSD. Recently, this technology as been made available in the form of a standard byte-addressable DDR4 DIMM on the CPU memory bus, just like DRAM DIMMs [25] with the new Cascade lake based chipsets and is called Optane DC PMM (different from Optane SSD). Optane DC PMMs are higher capacity than DRAM modules with up to 512 GB per module available today.

There are two operating modes for Optane DC PMM. In **2 Level Mode** (2LM or *cached*) PMM act as system memory with DRAM as a direct mapped cache. This operating mode allows for transparent use of the PMM at the overhead of maintaining a DRAM cache. **App Direct Mode** allows users manage the PMM directly. The PMM are mounted on a system as direct access file systems. Files on the PMM devices are then memory mapped into an application. When using a direct access aware file system, loads and stores to addresses in this memory mapped file go directly to the underlying PMM. Note that in App Direct mode, the total available memory is the sum of DRAM and PMM while in 2LM only the PMM capacity is counted. In this work, we focus on using the PMM in App Direct mode, and make comparisons between our optimized data movement and 2LM.

Figure 3 shows the read, write, and copy bandwidth of DRAM and PMM on our test system with six interleaved 128 GB PMMs. The read, write, and copy operations were implemented by splitting a region of memory into contiguous blocks and assigning a thread to each chunk. AVX-512 streaming loads and stores were used to implement the copy operation as they provide significantly higher throughput between DRAM and PMM.

From Figure 3, we make the following observations about PMM: bandwidth is significantly lower than DRAM, read bandwidth scales with the number of threads, write bandwidth peaks at a low number of threads and diminishes with a higher number of threads, copy bandwidth from DRAM to PMM scales with the number of threads, copy bandwidth is chiefly limited by PMM write bandwidth, and there is significant read/write asymmetry. These findings agree with the performance evaluation of Optane PMM by other researchers [25].

The read/write asymmetry has implications on the performance of kernels with inputs and outputs in PMM or DRAM. Figure 4 demonstrates the performance impact on a single CONV kernel. We observe that when the input to the CONV kernel is in PMM and the output is in DRAM, the performance of the kernel is comparable to when both input and output are in DRAM. However, in the cases where the output is in PMM, the kernel runs over two times slower. Any system seeking an optimal runtime with a memory constraint must take these relative timings into consideration when making decision on where to assign data. In the next section, we will describe the details of AutoTM and how it manages these performance characteristics.

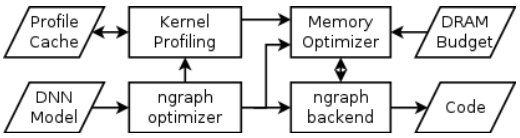


Figure 5. System Overview.

3 AutoTM

An overview of the proposed framework is shown in Figure 5. A DNN model is given to nGraph, which optimizes the network DAG according to the selected backend (e.g. CPU, GPU etc.). As part of the compilation process, our system inspects the nGraph DAG data structure to extract (1) the order and types nodes in the graph, (2) the tensors produced and consumed by each node, and (3) the specific kernels chosen by nGraph.

We then perform profiling on every kernel in the computation graph by varying its inputs and outputs between the different memory pools (i.e., DRAM or PMM) and recording the execution time of the kernel in each configuration. Since this step is potentially time consuming and DNNs typically contain many identical kernels, we keep a software cache of profiled kernels. By keeping a profile cache, profiling for a given DNN only needs to be performed once. Profiling and DAG information is then fed into a Memory Optimizer (described in Section 3.1) along with a DRAM capacity constraint, that mutates the nGraph data structure with the tensor assignments and data movement nodes.

A user of this system only needs a nGraph function, which is a collection of "Node" and "Tensor" data structures describing computations and data flow of the compute graph. These functions can be created by using one of the nGraph front ends, or directly using C++. Profiling, optimization, and code generation all happen as part of the nGraph compilation process and is transparent to the user.

In the rest of this section, we first give a high level introduction to the memory optimizer. Then we present the details of the optimizer's underlying ILP formulation.

3.1 Memory Optimizer

The goal of the Memory Optimizer is to *minimize* execution time by optimizing intermediate tensor movement and placement. The inputs to the optimizer are (1) the types of kernels in the computation graph in topological order, (2) the set of all valid tensor input/output locations for each kernel as well as profiled execution time for each configuration, (3) the sizes of all intermediate tensors, as well as their producers, users, and final users, (4) synchronous copy bandwidths between DRAM and PMM, and (5) a DRAM limit. The output of the optimizer describes the location and date movement schedules for all intermediate tensors that will minimize the global execution time of the graph.

Since the Memory Optimizer is implemented as an ILP, we need to model tensor location and movement using integer or binary variables and constraints [32]. For each tensor t, we create a separate network flow graph $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$ that traces the tensor's location during its lifetime. Examples of such graphs are given in Figure 6a and 6b. The structure of these graphs allows us to customize the semantics of possible tensor locations and movements.

Using this graph structure, we investigate two separate formulations, *static* and *synchronous*. The *static* formulation (Figure 6a) allows no tensor movement between memory pools. A tensor is assigned to either DRAM or PMM and remains there through its lifetime. The *synchronous* formulation (Figure 6b) allows tensors to be moved between memory pools but blocks program execution to perform this movement. We further generalize the ILP formulation to an asynchronous formulation that allows overlap between computation and data movement in Section 7.

Network flow constraints [16] are placed on each tensor flow graph $\mathcal{G}_t$ so that flow out of the source vertex is 1, flow into the sink vertex is 1, and flow is conserved for each intermediate node. The solution to this network flow describes the movement of the tensor. For example, the bold path in Figure 6b implies the following schedule for tensor t_1: (1) created by kernel k_1 in DRAM, (2) remains in DRAM until the execution of kernel k_2, (3) after k_2, synchronously moved t_1 into PMM, (4) prefetch t_1 into DRAM right before k_4, (5) move t_1 out of DRAM after k_4, (6) tensor t_1 is in PMM for the execution of kernel k_5, (7) after k_5, tensor t_1 is no longer needed and can be freed from all memory pools.

3.2 Objective Function

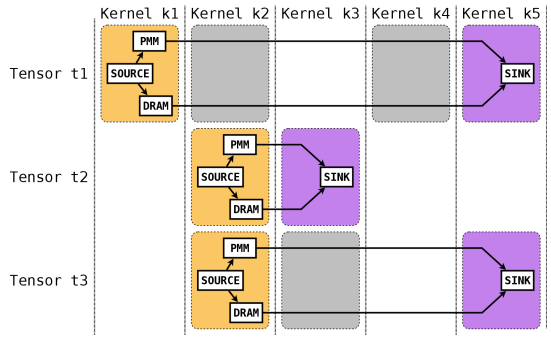
We wish to *minimize* the execution time of the computation graph under a DRAM constraint. In our framework, computation kernels are executed sequentially. Therefore, in the *static* formulation where there is no tensor movement, the objective function (expected execution time) is

$$\min \sum_{k \in \mathcal{K}} \rho_k \tag{1}$$

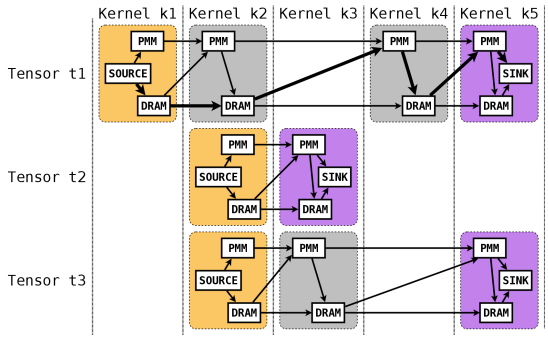
where $\mathcal{K}$ is the set of all kernels k in the computation graph and ρ_k is the expected execution time for kernel k. Note that ρ_k depends on the locations input and output tensor for kernel k. The selection of input and output tensor locations is not trivial because of dependencies between kernels. For example, if tensor t_3 in Figure 2 is assigned to PMM, then kernel k_2 must produce t_3 into PMM and kernels k_3 and k_4 must reference t_3 in PMM, which has a performance impact.

Given the lower performance of PMM relative to DRAM, the cost of moving a tensor from DRAM to PMM may be amortized by a resulting faster kernel execution. In the *synchronous* formulation, tensor movement that blocks computation graph execution and may only happen between kernel executions. The objective function then becomes

$$\min \sum_{k \in \mathcal{K}} \rho_k + \sum_{t \in \mathcal{T}} M_t^{\text{sync}} \tag{2}$$



(a) Static assignment. Tensors are created into either PMM or DRAM and stay there.



(b) Synchronous Movement. Tensors are allowed to move just before or just after the kernels that use or produce them. This movement blocks program execution.

Figure 6. Overlap of multiple tensor graphs and their interactions with kernels. Following the color coordination in Figure 2, **orange** denotes the producer of a tensor, **lilac** is the last user, **gray** marks the user of a tensor. We define the term **component** to refer to the subgraphs within each shaded region.

where $\mathcal{T}$ is the set of all intermediate tensors t in the computation graph and M_t^{sync} is the total amount of time spent moving tensor t. Note that a tensor t may be moved multiple times during its lifetime, so M_t^{sync} represents the sum of movement times of all individual moves of t.

3.3 DRAM Variables

As noted above, the execution time of a kernel depends on the locations of its input and output tensors. We must also keep track of all live tensors in DRAM to establish a constraint on the amount of DRAM used. Thus, we need machinery to describe for each kernel k whether the input and output tensors of k are in DRAM or PMEM and which tensors are in DRAM during the execution of k.

For each kernel $k \in \mathcal{K}$ and for each tensor $t \in \mathcal{T}$ where t is an input or output of k, we introduce a binary variable

$$t_{t,k}^{\text{DRAM}} = \begin{cases} 1 & \text{if } t \text{ is in DRAM } \textbf{during } k \\ 0 & \text{if } t \text{ is in PMM } \textbf{during } k \end{cases} \tag{3}$$

In practice, this variable is implemented as $t_{t,k}^{\text{DRAM}} = 1$ if and only if **any** of the incoming edges to the DRAM node in the component in the network flow graph $\mathcal{G}_t$ for k are taken.

To determine tensor liveness, we introduce binary variables

$$t_{t,k+}^{\text{DRAM}} = \begin{cases} 1 & \text{if } t \text{ is in DRAM } \textbf{after} \text{ kernel } k \\ 0 & \text{if } t \text{ is in PMM } \textbf{after} \text{ kernel } k \end{cases} \tag{4}$$

for each kernel $k \in \mathcal{K}$ and for each tensor $t \in \mathcal{T}$ where t is an output or output of k. These variables describe whether a tensor is written into DRAM after the execution of a kernel, and if it remains in DRAM until the next time it is used. In practice, this is implemented as $t_{t,k}^{\text{DRAM}} = 1$ if and only if the outgoing DRAM to DRAM edge is taken from the DRAM node in the component in network flow graph $\mathcal{G}_t$ for k.

We make these two distinct class of variables to handle the case in the *synchronous* formulation where a tensor is prefetched from PMM to DRAM as an input to some kernel k and then moved back to PMM immediately after k.

3.4 DRAM Constraints

Our main goal here is to establish a constraint on the amount of DRAM used by the computation graph. We must ensure that the sum of sizes of all live tensors in DRAM at any point is less than some limit $\mathcal{L}_{\text{DRAM}}$

We use the DRAM variables discussed in the previous section. First, define a helper function $\text{ref}(k, t) = k'$ where $k, k' \in \mathcal{K}$ and $t \in \mathcal{T}$ with k' defined as latest executing kernel earlier or equal to k in the topological order of the computation graph such that there exists DRAM node in $\mathcal{G}_t$ for kernel k'. For example, in Figure 6a, $\text{ref}(k_3, t_1) = k_1$ and in Figure 6b, $\text{ref}(k_3, t_1) = k_2$.

We want to ensure that at the execution time for each kernel $k \in \mathcal{K}$, the cumulative size of all live tensors resident in DRAM is with some limit $\mathcal{L}_{\text{DRAM}}$. Using the ref function, we add the following constraint for each $k \in \mathcal{K}$:

$$\sum_{t \in \mathbb{IO}(k)} |t| t_{t,k}^{\text{DRAM}} + \sum_{t \in \mathbb{L}(k)} |t| t_{t,\text{ref}(k)+}^{\text{DRAM}} \leq \mathcal{L}_{\text{DRAM},k} \tag{5}$$

where $|t|$ is the allocation size of tensor t in bytes, $\mathbb{IO}(k)$ is the set of input and output tensors for k, and $\mathbb{L}(k)$ is the set of all non-input and non-output tensors that are "live" during the execution of k. We assign a separate limit $\mathcal{L}_{\text{DRAM},k}$ for each kernel k initialized to $\mathcal{L}_{\text{DRAM}}$ to address the memory fragmentation issue discussed in Section 4.2

3.5 Kernel Configurations and Kernel Timing

For each kernel $k \in \mathcal{K}$, we use an integer variable ρ_k for the expected execution time of k given the locations of its input and output tensors. First, we define a configuration c

as a valid assignment of each of a kernel's input and output tensors into DRAM or PMM. For example, a kernel with one input and one output tensor may have up to four configurations, consisting of all combinations of its input and output in DRAM or PMM.

The definition of ρ_k is then

$$\rho_k = \sum_{c \in C(k)} n_{k,c} d_{k,c} \tag{6}$$

where $C(k)$ is the set of all valid configurations c for kernel k, $n_{k,c}$ is the profiled execution time of kernel k in configuration c, and $d_{k,c}$ is a one-hot indicator with $d_{k,c} = 1$ if and only if kernel k's input and output tensors are in configuration c.

3.6 Tensor Movement Timing

The movement cost of a tensor t is the size of the tensor $|t|$ divided by bandwidth between memory pools. Since bandwidth may be asymmetric, we measure and apply each separately. For each tensor $t \in \mathcal{T}$, the total synchronous movement time M_t^{sync} is the sum of the number of taken edges in $\mathcal{G}_t$ from DRAM to PMM multiplied by the DRAM to PMM bandwidth and the number of taken synchronous edges from PMM to DRAM multiplied by the PMM to DRAM bandwidth.

In our case where tensors are immutable, we may apply an optimization of only producing or moving a tensor into PMM once. Any future movements of this tensor into DRAM references the data that is already stored in PMM. Further movements from DRAM to PMM become no-ops.

4 Implementation Details

In this section, we describe some of the implementation details which are not directly part of the ILP formulation. The memory optimizer itself was implemented in the Julia [5] programming language using the JuMP [12] package for ILP modeling. Gurobi [18] was used as the backend ILP solver. We chose nGraph [11] over other popular machine learning frameworks based on static computation graphs as our backend because it is optimized for the Intel hardware and is relatively easy to modify. However, AutoTM is a general technique that can be integrated into other frameworks with similar underlying semantics.[1]

4.1 nGraph Compiler Backend

The nGraph compiler is an optimizing graph compiler and runtime developed by Nervana Systems/Intel for deep learning (DL) applications aiming to provide an intermediate representation (IR) between DL frameworks and hardware backends. An nGraph IR is a directed acyclic graph (DAG) of stateless operations nodes, each node with zero or more inputs, outputs, and constant attributes. Inputs and outputs of each node are multidimensional arrays called tensors with an arbitrary layout. The backend kernel used to implement a node is chosen based on the attributes of the node as well as the sizes, data types, and layouts of each of its inputs and outputs. nGraph will also apply generic and backend specific whole graph optimizations such as kernel fusion and algebraic simplification.

Memory location for intermediate tensors is performed using ahead-of-time heap allocation by traversing the function DAG and maintaining a list of live tensors. When tensors are last used, the memory space occupied by those tensors is freed and used for future tensors.

4.2 Managing Memory Fragmentation

The ILP formulation presented thus far assumes perfect memory management, which means that if the sum of sizes of live tensors is under the memory limit, then all tensors *will* fit within memory. In practice, this is not always the case. The process of allocating and freeing tensors may fragment memory resulting in a larger memory requirement.

To manage this, we use an iterative process of reducing the DRAM limit for kernels where the the following limit is exceed and rerunning the ILP.

1. We initialize the kernel-wise DRAM limits $\mathcal{L}_{\text{DRAM},k}$ to the $\mathcal{L}_{\text{DRAM}}$.
2. We solve the ILP using the current values of $\mathcal{L}_{\text{DRAM},\mathcal{K}}$. nGraph translates the resulting schedule and then executes its memory allocator pass.
3. We collect the set of kernels $\mathcal{K}_{\text{frag}}$ where the total amount of memory allocated exceeds $\mathcal{L}_{\text{DRAM}}$ due to fragmentation. If this set is empty, we are done.
4. Otherwise, we apply an update $\mathcal{L}_{\text{DRAM},k} = 0.98\mathcal{L}_{\text{DRAM},k}$ for all $k \in \mathcal{K}_{\text{frag}}$ and go back to step (2).

Thus, the ILP solver may have to run multiple times before a valid solution is found. In practice, this process is usually only done 1 to 2 times with a maximum of 5 as discussed in Section 6.6.

4.3 Data Movement Implementation

Synchronous movement operations are integrated as new *move* nodes in the nGraph compiler, which are automatically inserted into the nGraph computation graph following memory optimization. The implementation of these move nodes uses a multithreaded memory copy with AVX-512 streaming load and store intrinsics followed by a fence.

Operation scheduling in nGraph consists of a simple topological sort of the nodes in the computation graph, beginning with the input parameters. This creates unnecessary memory usage with move nodes as they are scheduled ad hoc, resulting in tensor lifetimes that are longer than necessary. Thus, we extended the nGraph scheduler so that if a tensor is moved from DRAM to PMM after some kernel k, we ensure that this movement occurs immediately *after* the execution of k. Conversely, if a tensor is moved from PMM to DRAM

[1] All of the AutoTM code can be found on GitHub at https://github.com/darchr/AutoTM.

to be used for kernel k, we ensure this occurs immediately *before* the execution of k.

5 Evaluation Methodology

5.1 System

Our experimental Optane DC system was a prototype dual socket Xeon Cascade-Lake server. Each socket had 6×32 GB of DRAM and 6×128 GB Intel Optane DIMMs. Each CPU had 24 hyperthreaded physical cores. In total, the system had 384 GB of DRAM and 1.5 TB NVDIMM storage.

NUMA policy was set to local by default. Unless specified otherwise, all experiments were conducted on a single socket with one thread per physical core. Each workload was run until execution time per iteration (traversal of the computation graph) was constant. Since these workloads contain no data dependent behavior, performance will be constant after the first couple of iterations. Checks were used to ensure no IEEE NaN or subnormal numbers occurred, which can have a significant impact on timing [3].

Our approach does not change the underlying computations performed during training; it is a transparent backend implementation optimization. Thus, the performance of our benchmarks across a few training iterations is sufficient to obtain performance metrics.

We chose to evaluate AutoTM with a multicore CPU platform because Optane PMMs are only available for CPU platforms. However, the ILP formulation of AutoTM should apply to any heterogeneous memory system. We explore one other example with CPU and GPU DRAM in Section 7.

5.2 DNN Benchmarks

We choose a selection of state of the art Deep Neural Networks for benchmarking our approach. A summary of the benchmarks and batch sizes used is given in Table 1. Conventional CNNs for the Optane DC system were Inception v4 [41], Resnet 200 [21], DenseNet 264 [24], and Vgg19 [39]. All but Vgg19 have complex dataflow patterns to stress test AutoTM. The batch sizes were chosen to provide a memory footprint of over 100 GB for each workload. These batch sizes, while larger than what is typically used, mimic future large networks while still fitting within the DRAM of a single CPU socket of our test system.

We also compare our approach against the native 2LM mode, which is a hardware solution to data management that uses PMM transparently with CPU DRAM as a cache. Since we can not change the physical amount of DRAM used by 2LM, we used very large neural networks that exceed the CPU DRAM and require the use of PMM to train. These very large networks include Vgg416 [36] (constructed by adding 20 additional convolution layers to each convolution block in Vgg16) and Inception v4 with a batch size of 6144.

Benchmark	Batchsize	System	Baseline Memory (GB)
Inception v4	1024	PMM	111
Resnet 200	512	PMM	132
Vgg 19	2048	PMM	143
DenseNet 264	512	PMM	115
Inception v4	6144	Large PMM	659
Vgg 416	128	Large PMM	658
Resnet 200	2560	Large PMM	651
DenseNet 264	3072	Large PMM	688
Inception v4	64, 128, 256	GPU	7.6, 14.7, 29.8
Resnet 200	32, 64, 128	GPU	8.7, 16.9, 32.2
DenseNet 264	32, 64, 128	GPU	8.5, 16.8, 32.4
Vgg 19	64, 128	GPU	7.1, 12.6

Table 1. Summary of the benchmarks used in this work.

5.3 Experiments

We want to determine whether PMM is cost effective for training DNNs, and how AutoTM compares against existing solutions to use PMM.

For the conventional benchmarks, we consider the impact of performance with different ratios between PMM and DRAM. These ratios are given in the form $a : b$ where a is the amount of PMM relative to b the amount of DRAM used to train the network. A ratio of $1 : 1$ indicates that a network was trained with half PMM and half DRAM. For a network requiring 128GB total to train would have a split of 64 GB PMM and 64 GB DRAM. Setting a ratio such as this may lead to a larger total memory footprint in total due to memory fragmentation in both PMM and DRAM. However, in practice the total memory footprint expansion is minimal with an observed maximum observed value of 3.83% occuring in the *static* formulation for Inception v4. A ratio of $0 : 1$ denotes a system where only DRAM is used while $1 : 0$ is a system using only PMM.

We use a baseline of a first-touch NUMA allocation policy with DRAM as a near node and PMM as a far node for the conventional benchmarks. The NUMA policy was encoded in our framework by assigning intermediate tensors to DRAM as they are created until the modeled memory capacity of DRAM is reached. Future tensors can only reside in DRAM if existing tensors are freed.

For the large benchmarks, we compare our approach to the 2LM hardware managed DRAM cache to determine the effectiveness of AutoTM relative to an existing approach.

6 Results

6.1 Conventional Networks

Figure 7 shows the speedup provided by our scheduling for over training solely with PMM (ratio $0 : 1$). The horizontal axis is the ratio of PMM to DRAM used to train the network. We observe that when PMM is used as a direct substitute for DRAM, performance is poor with a 3x to 8x increase in training time (red horizontal line). However, with a minimal amount of DRAM such as an $8 : 1$ PMM to DRAM ratio, we

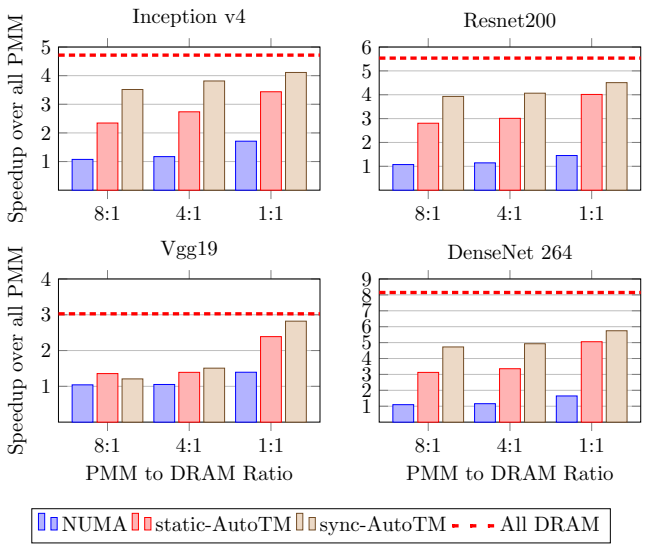


Figure 7. Results for conventional networks. Note different y-axes. The baseline (1.0 in the graphs) is a system with only PMM (ratio of 0 : 1).

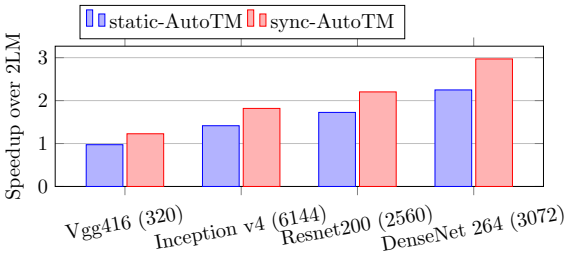


Figure 8. Performance of the static and synchronous formulations relative to 2LM cached mode.

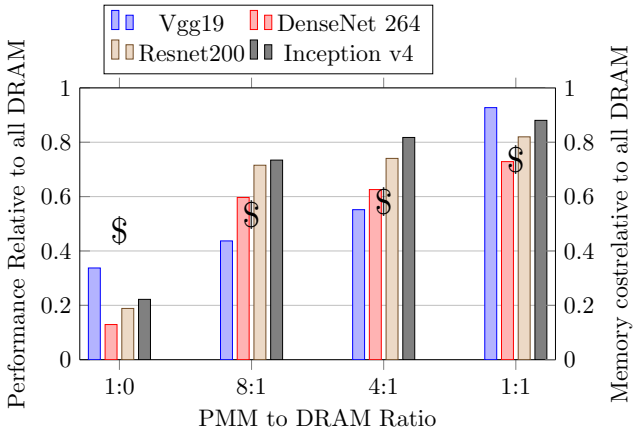


Figure 9. Price–performance analysis. The bars (left axis) show the network performance relative to all DRAM while the dollar signs (right axis) show the memory system price relative to all DRAM. The regions where the bars are higher than the dollar signs are regions where price–performance is lower.

are able to dramatically improve performance without changing the overall memory footprint of the application. Further, adding more DRAM only marginally increases performance.

This above performance gain does not occur using conventional first-touch NUMA. This is because first-touch NUMA [27] works by allocating tensors into DRAM as they are used by the computation graph until the DRAM capacity is reached. In the training of DNNs, tensors produced early on in the forwards pass are used during the backwards pass and thus must be live for the majority of the graph's computation [36]. With first-touch NUMA, these long lived tensors are assigned to DRAM forcing future short-lived tensors into PMM.

AutoTM, on the other hand, is aware of the performance implication of these long lived tensors. The general strategy AutoTM takes is to prioritize short-lived tensors for DRAM placement (Section 6.4). These short-lived tensors mainly include intermediate tensors generated during the backwards pass. By prioritizing short-lived tensors, AutoTM ensures that more tensors overall may reside in DRAM.

Vgg is an outlier due to its extremely large second convolution layer. With small DRAM sizes, some or all of the input and output tensors of this large layer must be placed in PMM, incurring a performance penalty. Once these tensors can be placed in DRAM, we see a significant performance improvement as can be seen in the performance jump from the 4 : 1 ratio to the 1 : 1 ratio. Another interesting feature of this network is that the *synchronous* formulation performs slightly worse than the *static* formulation for an 8 : 1 ratio. This is caused by the interaction between the insertion of move nodes and the defragmentation procedure.

6.2 Comparison to a hardware DRAM cache

We use very large networks to compare AutoTM to a hardware-controlled DRAM cache (2LM mode). The results from the large benchmarks Vgg416 and the large batchsize Inception v4 are shown in Figure 8. The *static* formulation has performance comparable to 2LM with the *synchronous* formulation running 23% faster. We see further improvement for the other networks, with Resnet 200 running over 2x faster than 2LM. Inception v4, on the other hand, runs almost 2x faster under the synchronous formulation than under 2LM. Since 2LM is a DRAM caching strategy, there is overhead involved in maintaining cache metadata. For complex networks like Resnet 200 and DenseNet, it is likely that 2LM is unable to perform accurate prediction and prefetching. This leads to a performance penalty for 2LM that is not incurred by AutoTM. Additionally, the 64-byte block-based data movement of the hardware DRAM cache may reduce performance compared to the large contiguous data movement in AutoTM.

6.3 Cost-Performance Analysis

Can PMM offer a cost performance advantage over DRAM for training large DNNs? Table 2 provides a summary of

Capacity (GB)	Price per DIMM	Price per GB
DRAM 8	$190.45	$23.81
DRAM 16	$265.82	$16.61
DRAM 32	$602.50	$18.83
DRAM 64	$1,255.75	$19.62
DRAM 128	$2,512.00	$19.63
Optane 128	$1,004.50	$7.85
Optane 256	$3,466.75	$13.54
Optane 512	$10,552.00	$20.61

Table 2. Lenovo price summary of Optane and server class DRAM. (see footnote 2)

module cost and cost per GB for a selection of server class DRAM and Optane DIMMs, as quoted by Lenovo[2]. The price per GB of DRAM stays roughly constant across module sizes. PMM, on the other hand, increases in price per GB as capacity increases. Prices are driven by business decisions. Because a 512 GB DRAM DIMM is not available, a premium can be charged for this capacity module.

For our analysis, we use the price of the cheapest PMM at $7.85 per GB and the cheapest DRAM at $16.61 per GB. This means the cost-per-GB advantage of PMM over DRAM is about 2.1x. In Figure 9 we only include the cost of the memory actually used. Since Optane DC is a new technology, prices are still adjusting, and as the technology matures, price will likely decrease, improving its cost-effectiveness.

Figure 9 shows the relative performance of AutoTM for our workloads (bars, left axis) as well as the cost of memory used by the application relative to the case where all DRAM is used (dollars, right axis). The use of PMM can be cost effective if the performace lost by replacing some DRAM with PMM is less than the cost reduction. We observe that only using PMM directly is not cost effective, the performance loss caused by the slower devices is not offset by the lower price. However, for PMM to DRAM ratios of $4:1$ and $1:1$, AutoTM can provide a cost-performance benefit. This cost-performance benefit may be reduced when taking the whole system into account, but the cost of memory is usually the dominant cost in large systems.

6.4 Understanding the ILP Solution

In this section, we present some insight to how and why AutoTM works using Figure 10. Figure 10a shows the slowdown of the *static* and *synchronous* relative to all DRAM. With a small amount of DRAM, performance improves rapidly. This trend continues until a critical threshold where adding DRAM yields diminishing returns.

To understand this behavior, we look at the input and output memory locations for each kernel as well as the amount of data moved. Figure 10b shows the percent by memory footprint of kernel input and output tensors in DRAM. We see a trend to assign as many kernel inputs and outputs into DRAM, with a slight priority on output tensors. This is consistent with the lower write bandwidth of PMM. Furthermore, the point where almost 100% of output/input tensors are in DRAM corresponds to the critical point in the performance graphs. This implies a general strategy to maximize kernel read and write memory accesses in DRAM, followed by data movement to PMM when DRAM capacity constrained.

This idea is reinforced by Figure 10c, which shows the total amount of memory moved between DRAM and PMM in the *synchronous* formulation. With a DRAM limit near zero, no data movement occurs since no data may be moved into DRAM. A small DRAM allowance, however, is followed by a dramatic increase in data movement, again with an emphasis on moving data from DRAM to PMM. Once the DRAM limit allows almost all tensor inputs/outputs to reside in DRAM, the amount of data movement decreases. The region of gradual slowdown seen in the performance plot is caused primarily by data movement rather than kernel slowdown from more memory accesses to PMM.

6.5 Kernel Profiling Accuracy

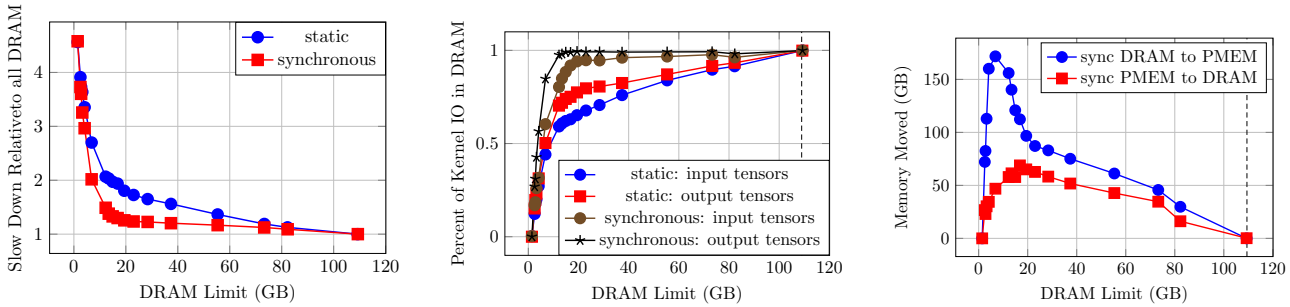
To evaluate the accuracy of our profile based approach, we show the error between the expected runtime and the measured runtime in Figure 11. The worst case error occurs for in the *static* formulations for DenseNet 264 (19%). This error is likely due to CPU caching. During profiling, move nodes are placed at the inputs of kernels under test to allow the inputs and outputs of the kernel to be varied between DRAM and PMM. Kernels cannot be directly profiled due to levels of indirection used in nGraph. Because move nodes are implemented using streaming instructions, no data is resident in CPU caches following these instructions. Hence, our profiling step is essentially measuring the cold-performance of these kernels. This results in an *overestimation* in run time for the *static* formulation since no move nodes are used. Vgg19 is less affected due to its very large intermediate layers.

The expected runtime for the *synchronous* formulation closely follow the predicted runtime because of the use of move nodes placed in the computation graph. The error in the $1:0$ all PMM case exists for similar reasons.

6.6 ILP Solution Times

It is important that the memory optimizer is able to run in a reasonable amount of time. Although ILP is inherently *NP*-hard, recent solvers can find solutions to many problems quickly. Table 3 shows the total amount of time optimizing the ILP. The number of retries due to memory fragmentation is shown in parentheses. Solution time increases with model complexity. Since the optimized computation graph will run for days or weeks to fully train the DNN, this optimization overhead will be amortized. The worst case is the static

[2]https://www.lenovo.com/us/en/p/7X05A01TNA/customize?dcscGuid=f3dd16d0-96dd-4deb-9c48-9c6cec9578ba (accessed August 14, 2019)



(a) Slowdown relative to all DRAM.

(b) Percent by memory size of all kernel inputs and outputs in DRAM.

(c) Amount of data moved between DRAM and PMM.

Figure 10. AutoTM's solution strategy for Inception v4.

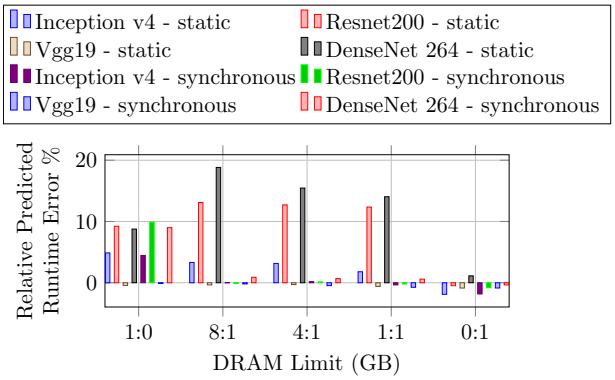


Figure 11. Comparison of actual execution time and execution time predicted by kernel-wise profiling for the conventional networks.

	Static			Synchronous		
Network	8 : 1	4 : 1	1 : 1	8 : 1	4 : 1	1 : 1
Vgg19	0.40 (1)	0.70 (2)	0.82 (2)	2.5 (5)	1.7 (3)	0.94 (2)
Inception v4	37.9 (5)	16.4 (2)	13.7 (2)	50.3 (6)	15.3 (2)	16.4 (2)
Resnet 200	2846 (2)	3105 (3)	91.9 (1)	710 (2)	571 (2)	79.9 (2)
DenseNet	3307 (1)	2727 (1)	2582 (1)	1448 (3)	2021 (3)	1404 (2)

Table 3. Gurobi ILP solver time to a relative MIP gap of 0.01 for the *static* and *synchronous* formulations for the conventional networks. Entries of the form a (b) indicate the total time a in **seconds** it took to solve the ILP b times. Multiple solutions are needed in the case of memory fragmentation management.

formulation for DenseNet which takes a little less than an hour to fully solve.

7 Extending AutoTM

In this section, we discuss two extensions to AutoTM: allowing *asynchronous* data movement and performing kernel implementation selection. We explain why these extensions were not included in the original formulation and demonstrate their viability on a CPU-GPU platform. These extensions and the GPU implementation of AutoTM show that it is a general and flexible framework for managing heterogeneous memory.

The first extension we investigate is *asynchronous* offloading and prefetching of intermediate tensors between memory pools. This allows data movement to be overlapped with computation, improving the throughput of the application as a whole. We implemented asynchronous data movement on the PMM system, but found it performed poorly on existing CPU only systems for a number of reasons. Neither a dedicated copy thread nor DMA provided sufficient performance to mitigate the overhead of these approaches. However, a PCIe connected GPU offers a high speed asynchronous data copy API, which is ideal for implementing this extension.

The second extension to the formulation is performing kernel implementation selection. The underlying library used by nGraph to perform forward and backward convolutions for the GPU backend is cuDNN [9], a deep learning library from Nvidia. This library exposes several different implementations for each convolution, each with performance and memory footprint tradeoffs. Generally, faster implementations require more memory. In a memory starved case, this larger memory footprint may require more offloading of previous tensors, resulting in a global slowdown. Since nGraph does not expose any kernel selection options for the CPU backend, we implement this on the GPU instead.

7.1 ILP Formulation Modifications

Since AutoTM is implemented using an ILP formulation, we can extend it to be aware of the performance and memory footprint of these different kernels and globally optimize tensor movement and implementation selection. Here, we provide a high level overview of the additions to the ILP formulation to express asynchronous data movement and kernel implementation selection.

7.1.1 Objective Function:

In our formulation, we allow an arbitrary number of tensors to be moved between GPU and CPU DRAM concurrently

with a single kernel. This results in a new objectives function

$$\min \sum_{k \in \mathcal{K}} \max \left\{ \rho_k, \sum_{t \in \text{ASYNC}(k)} M^{\text{async}}_{t,k} \right\} + \sum_{t \in \mathcal{T}} M^{\text{sync}}_t \quad (7)$$

where $\text{ASYNC}(k) = \{t \in \mathcal{T} : t \text{ can be move concurrently with } k\}$ and $M^{\text{async}}_{t,k}$ is the amount of time (if any) spent moving tensor t during the execution of k. The max operation is implemented using standard ILP techniques.

7.1.2 Tensor Graphs:

We must extend the tensor flow graphs $\mathcal{G}_t$ to encode points of asynchronous tensor movement. We identify kernels that can be overlapped with data movement and add a component in each tensor's graph (like those shown in Figure 6b) for each kernel with which the tensor can be moved concurrently.

7.1.3 Asynchronous Data Movement:

Asynchronous move times for tensor t must be generated for each kernel k across which t may be moved. This comes directly from the extended tensor graph

$$M^{\text{async}}_{t,k} = \left(\frac{|t|}{\text{BW}^{\text{ASYNC}}_{P \to D}} \right) e_{P \to D} + \left(\frac{|t|}{\text{BW}^{\text{ASYNC}}_{D \to P}} \right) e_{D \to P} \quad (8)$$

where $e_{P \to D}$ ($e_{D \to P}$) is the binary edge variable in $\mathcal{E}_t$ corresponding to the asynchronous movement of t from PMM to DRAM (DRAM to PMM) across kernel k.

7.1.4 Selecting Kernel Implementations:

Let $\mathcal{I}(k) = \{1, 2, \ldots, n_k\}$ be an enumeration of the implementations for kernel k. We generate one-hot binary variables $v_{i,k}$ for all $i \in \mathcal{I}(k)$ where $v_{i,k} = 1$ implies implementation i is to be used for kernel k.

7.1.5 DRAM Constraints:

Constraining DRAM is similar to the *static* and *synchronous* formulations, but now includes kernel memory footprints with

$$\sum_{i \in \mathcal{I}(k)} s_{k,i} v_{k,i} + \sum_{t \in \mathbb{IO}(k)} t^{\text{DRAM}}_{t,k} + \sum_{t \in \mathbb{L}(k)} t^{\text{DRAM}}_{t,\text{ref}(k)+} \leq \mathcal{L}_{\text{DRAM}} \quad (9)$$

where $s_{k,i}$ is the memory footprint of implementation i of k.

7.1.6 Kernel Timing:

The expected runtime of a kernel is now dependent on which implementation of the kernel is chosen. Building on the example given in Section 3.5, assume that k has two implementations (i.e. $\mathcal{I}(k) = \{1, 2\}$). The expected execution time for ρ_k kernel k is modeled as

$$\rho_k = \sum_{c \in \mathcal{C}(k)} \sum_{i \in \mathcal{I}(k)} n_{k,c,i} (d_{k,c} \wedge v_{i,k}) \quad (10)$$

with $n_{k,c,i}$ is the profiled runtime of implementation i of kernel k in IO configuration c. This approach does not account for the performance impact of memory conflict between data movement and the computation kernel. However, the maximum memory bandwidth of our GPU is 616 GB/s while the maximum bandwidth of PCIe is 16 GB/s. Thus, the impact of asynchronous data movement is likely low.

7.2 Implementation

We modified the GPU backend of nGraph to support synchronous and asynchronous tensor movement as well as to allow for kernel selection of forward and backward convolution kernels. All GPU kernels are profiled with inputs and outputs in GPU memory. When implementation selection is available, all possible implementations of a kernel are profiled as well. Asynchronous movement was implemented using two CUDA [31] streams: one for computation and the other for data movement via *cudaMemcpyAsync*. These streams are synchronized before and after an asynchronous movement/computation overlap to ensure data integrity.

7.3 Methodology

Our system used a Nvidia RTX 2080 Ti with 11 GB of GDDR6 using CUDA 10.1 and cuDNN 7.6. The host system was an Intel Core i9-9900X with 64 GB of DDR4 DRAM.

We use the same convolutional neural networks used earlier. The networks and batch sizes used are given in Table 1. We compare the results of AutoTM with the performance of *cudaMallocManaged*, which is a memory virtualization layer offered by Nvidia for automatically moving data from the CPU to the GPU in the event of a GPU page fault and moving unused pages from GPU DRAM to CPU DRAM.

7.4 GPU Results

The results for the GPU experiments are given in Figure 12. For networks that fit on the GPU, our approach has no overhead as the ILP optimizer realizes no data movement is needed. As the intermediate working set increases, we observe a several fold improvement with AutoTM over *cudaMallocManaged* due to the lack of runtime overhead of our approach and its algorithm awareness. AutoTM provides considerable speedup when data movement between the CPU and GPU is required. The *asynchronous* extension outperforms the *synchronous* formulation with its ability to overlap data movement and computation. However, the *asynchronous* extension is limited to overlapping tensor movement with a single kernel at a time. Since the RTX 2080 Ti executes kernels faster than data movement, time must be spent to synchronize the two CUDA streams.

The synchronization overhead of overlapping tensor movement with a single kernel can be seen by comparing the achieved performance with the theoretical best performance, calculated by assuming infinite GPU DRAM capacity and using the fastest possible implementations for all kernels. As the memory requirement for training increases, AutoTM achieves a lower fraction of this best performance due to synchronization.

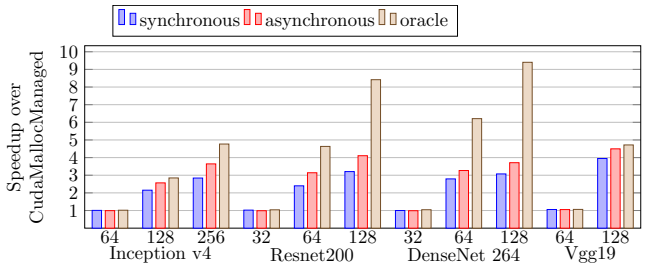


Figure 12. GPU performance of AutoTM relative to *cudaMalloc-Managed*.

We did not compare our results directly against vDNN [36] for two reasons. First, the RTX 2080 Ti GPU is much faster than the Titan X used in that work and thus we cannot compare results directly. Second, the code for vDNN is not available, making direct testing on our GPU difficult. However, while vDNN leverages the same characteristics as AutoTM (communication overlapping, kernel selection, and liveness analysis), AutoTM uses mathematical optimization rather than heuristics providing a more general solution.

8 Related Work

As an emerging technology Intel Optane DC has been explored in several recent works. These include in depth performance analysis [25], large graph analytics [15], and database I/O primitives [42]. Research into using Optane PMM for virtual machines demonstrates that only a small amount of DRAM is needed [23]. Flash based SSDs have also been used to reduce the DRAM footprint in database [13] and ML [14] workloads. These approaches use a software managed DRAM cache to mitigate the slow performance and block level read/write granularity of NVM SSDs. Operating system support for managing heterogeneous memory [2, 45] and support for transparent unified memory between GPU and CPU [26, 33] have been studied extensively in the past. However, to the best of our knowledge, the proposed work is the first to explore the design space and cost–performance tradeoffs of large scale DNN training on systems with DRAM and PMM.

Previous works such as vDNN [36] exploit heterogeneous memory between GPUs and CPUs by recognizing that the structure of DNN training computation graphs has a pattern where intermediate tensors produced by early layers are not consumed until much later in the graph execution. The authors of vDNN exploit this to develop heuristics for moving these tensors between GPU and CPU DRAM during training to free GPU memory. SuperNeurons [44] and moDNN [8] build on vDNN. SuperNeurons introduces a runtime manager for offloading and prefetching tensors between GPU and CPU memory as well as a cost-aware method of applying recomputation of forward pass layers during the backward pass to reduce memory. Similar to our approach, moDNN allows tensors to be offloaded and uses profiling information of kernel runtime and expected transfer time to determine how it will overlap computation and communication. AutoTM differs from these previous approaches in that we use mathematical optimization rather than problem specific heuristics. AutoTM also generalizes the location of data across DRAM and PMM instead of requiring data to be in DRAM for computation.

Integer Linear Programming and profile guided optimization have been used widely to address similar problems in research literature. For example, work in the embedded system space [4] uses ILP in to optimize the allocation of heap and stack data between fast SRAM and slow DRAM. ILP has also been used in register allocation [17] and automatic program parallelization [20]. ILP has been used to optimize instruction set customization and spatial architecture scheduling [32]. Profile guided optimization has been used for dynamic binary parallelization [48], process placement on SMP clusters [7] and online autotuning of CPU and GPU algorithm selection [34]. AutoTM builds on these ideas to address the new problem of data movement in heterogeneous memory systems.

9 Conclusions

We present AutoTM, an ILP formulation for modeling and optimizing data location and movement in static computation graphs such as those used for training and inference of DNNs. AutoTM uses profile data to optimally assign kernel inputs and outputs into different memory pools and schedule data movement between the two pools to minimize execution time under a memory constraint. With AutoTM, we can obtain 2x performance improvement over hardware DRAM caching solutions. We further find Intel Optane PMM can reduce the DRAM footprint of DNN training by 50 to 80% without significant loss in performance. Given the lower cost of Optane PMM, this can yield a cost-performance benefit in systems with mixed DRAM and PMM over a system with only DRAM.

AutoTM uses minimal problem specific heuristics, making it generally applicable to different systems and networks. We demonstrate this flexibility by extending AutoTM to GPUs, and believe it can be further extended to further heterogeneous systems, such as those with multiple GPUs or multi-level systems with HBM, DRAM, and PMM.

10 Acknowledgements

This work is supported in part by the Intel corporation and by the National Science Foundation under Grant No. CNS-1850566.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan,

Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.

[2] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In Yunji Chen, Olivier Temam, and John Carter, editors, *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*, pages 631–644. ACM, 2017.

[3] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 623–639, Washington, DC, USA, 2015. IEEE Computer Society.

[4] Oren Avissar, Rajeev Barua, and Dave Stewart. Heterogeneous memory management for embedded systems. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '01, pages 34–43, New York, NY, USA, 2001. ACM.

[5] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607, 2014.

[6] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[7] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In Gregory K. Egan and Yoichi Muraoka, editors, *Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, Cairns, Queensland, Australia, June 28 - July 01, 2006*, pages 353–360. ACM, 2006.

[8] X. Chen, D. Z. Chen, and X. S. Hu. modnn: Memory optimal dnn training on gpus. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 13–18, March 2018.

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[10] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[11] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *CoRR*, abs/1801.08058, 2018.

[12] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

[13] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim M. Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM footprint with NVM in facebook. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 42:1–42:13, 2018.

[14] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. *CoRR*, abs/1811.05922, 2018.

[15] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane DC persistent memory. *CoRR*, abs/1904.07162, 2019.

[16] Andrew Goldberg, Éva Tardos, and Robert Tarjan. Network flow algorithms. page 80, 04 1989.

[17] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0–1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, August 1996.

[18] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2018.

[19] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3D XPoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[20] Mary W. Hall, Jennifer-Ann M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Digital Technical Journal*, 10(1):71–80, 1998.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[22] Joel Hestness, Newsha Ardalani, and Gregory Diamos. Beyond human-level accuracy: Computational challenges in deep learning. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 1–14, New York, NY, USA, 2019. ACM.

[23] Takahiro Hirofuchi and Ryousei Takano. The preliminary evaluation of a hypervisor-based virtualization mechanism for intel optane DC persistent memory module. *CoRR*, abs/1907.12014, 2019.

[24] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[25] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[26] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 142–151. ACM, 2011.

[27] Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40:40–40:51, July 2013.

[28] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[30] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[31] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[32] Tony Nowatzki, Michael Ferris, Karthikeyan Sankaralingam, Cristian Estan, Nilay Vaish, and David Wood. Optimization and mathematical modeling in computer architecture. *Synthesis Lectures on Computer Architecture*, 8(4):1–144, 2013.

[33] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. Fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme. In Pen-Chung Yew, Sangyeun Cho, Luiz DeRose, and David J. Lilja, editors, *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, pages 33–42. ACM, 2012.

[34] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable performance on heterogeneous architectures. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 431–444, New York, NY, USA, 2013. ACM.

[35] Milan Radulovic, Darko Zivanovic, Daniel Ruiz, Bronis R. de Supinski, Sally A. McKee, Petar Radojković, and Eduard Ayguadé. Another trip to the wall: How much will stacked dram benefit hpc? In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pages 31–36, New York, NY, USA, 2015. ACM.

[36] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 18:1–18:13, Piscataway, NJ, USA, 2016. IEEE Press.

[37] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[38] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017.

[39] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[40] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[41] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[42] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory I/O primitives. *CoRR*, abs/1904.01614, 2019.

[43] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, et al. Alphastar: Mastering the real-time strategy game starcraft ii. *DeepMind Blog*, 2019.

[44] 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. *CoRR*, abs/1801.04380, 2018.

[45] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 331–345. ACM, 2019.

[46] Heiga Ze, Andrew Senior, and Mike Schuster. Statistical parametric speech synthesis using deep neural networks. In *2013 ieee international conference on acoustics, speech and signal processing*, pages 7962–7966. IEEE, 2013.

[47] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris N Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5907–5915, 2017.

[48] Ruoyu Zhou and Timothy M. Jones. Janus: Statically-driven and profile-guided automatic dynamic binary parallelisation. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 15–25, Piscataway, NJ, USA, 2019. IEEE Press.

A Artifact Appendix

A.1 Abstract

Our artifact contains the full source code created by the authors for this work. It includes the changes made to the ngraph compiler, a ngraph front-end for creating neural networks, the ILP modeling code, and the code for running experiments and generating the plots in this work. The code is capable of running on either a Intel Optane DC PMM equipped system or a system with an Nvidia GPU.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Integer Linear Program for modeling data-movement for static computation graphs.
- **Compilation:** Scripts for compilation are included. Clang 6.0 or newer required.
- **Run-time environment:** Ubuntu 18.04 with Linux Kernel v4.2 or newer. Root access required if using a Optane DC PMM equipped system to set up and mount NVDIMMs. If running GPU experiments, CUDA 10 and cuDNN 7.6.
- **Hardware:** For the Optane DC experiments, a 2-socket Intel Cascade Lake server with Optane DC support. For the GPU experiments, an Nvidia GPU is needed.
- **Run-time state:** Execution of the benchmarks on a Optane DC equipped system should be performed shortly after reboot to minimize memory fragmentation.
- **Execution:** For the Optane DC equipped system, process pinning to happens automatically.
- **Metrics:** Execution time, tensor location metrics, and profiling error.
- **Output:** Serialized Julia dictionaries holding intermediate results, LaTeX source/PDFs for plots based on the result data.
- **Experiments:** Manual invocation of high level functions.
- **How much disk space required (approximately)?:** 5-10 GB of disk space after compilation.
- **How much time is needed to prepare workflow (approximately)?:** 20-30 minutes for compilation of dependencies.
- **How much time is needed to complete experiments (approximately)?:** Optane DC system: Conventional workloads take 1-4 hours each to profile and benchmark. Large workloads can take between 4 and 12 hours. Small test workloads take several minutes.
  GPU system: 3-4 hours to repeat all experiments.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT License
- **Archived (provide DOI)?:** 10.5281/zenodo.3612698

A.3 Description

A.3.1 How delivered

Full AutoTM source code, benchmarks, and scripts used for this work are available under the DOI: `10.5281/zenodo.3612698`. The development version of the project is available on GitHub at https://github.com/darchr/AutoTM. The code is open sourced under the MIT license.

A.3.2 Hardware dependencies

Optane DC: A 2 socket Cascade Lake server system with support for Optane DC PMM is required. Furthermore, this work uses the (2-2-2) setup where each memory channel contains one DRAM DIMM and one NVDIMM.

GPU: An Nvidia GPU is required. The GPU used in this paper was an RTX 2080Ti.

A.3.3 Software dependencies

Both the CPU and GPU portions of this paper used Ubuntu 18.04 as the operating system. AutoTM requires Julia 1.2 or later, Clang 6.0, and all the dependencies of the ngraph compiler. The commercial Gurobi ILP solver versions 8 or 9 are necessary for full reproducibility, but is not necessary for basic functionality.

If using on an Optane DC equipped system, Linux Kernel v4.2 or newer is required for direct-access (dax) filesystem support. If using an Nvidia GPU system, CUDA 10 and cuDNN 7.6.

A full list can be viewed at http://arch.cs.ucdavis.edu/AutoTM/dev/software/.

A.4 Installation

Detailed instructions are provided in `/docs/src/installation.md` in the artifact directory or on GitHub at: http://arch.cs.ucdavis.edu/AutoTM/dev/installation/. Furthermore, an example Dockerfile that will successfully build the code for the GPU portion of the project can be found in the 'docker/' directory of the artifact or on Github at https://github.com/darchr/AutoTM/blob/master/docker/gpu.

The Dockerfile serves to demonstrate the base functionality of the artifact. For full results reproduction, the Gurobi ILP solver is necessary as the open source Cbc solver that is installed automatically as part of the build process is not powerful enough to solve the larger ILP problems. However, since Gurobi is a commercial tool, a license is required which does not allow the software to run inside of a container. Hence, the artifact must be installed directly on the test machine with the Gurobi software (see artifact installation instructions). Note that Gurobi provides single machine academic licenses for free.

A.5 Experiment Workflow

Workflow involves invoking top-level functions from the Julia REPL (read-eval-print loop). The steps to reproduce Figures 7-12 of the paper are provided in the documentation in the artifact repository: http://arch.cs.ucdavis.edu/AutoTM/dev/workflow/.

A.6 Evaluation and expected result

For conventional networks, expected results are runtime predicted by the ILP, actual runtime of the network, the DRAM limit passed to the optimizer, the elapsed time for model optimization, the size of data allocated to the near memory pool, and the size of data allocated to the far memory pool.

Further metrics include statistics on the number and types of Move nodes generated, the amount of data moved between memory pools, and the pool assignments of all intermediate tensors.

A complete list of gathered metrics is available here: https://arch.cs.ucdavis.edu/AutoTM/dev/results.

A.7 Experiment Customization

The artifact contains several methods for customization. For the CPU portion, the number of threads used for kernel implementations is configurable as well as the PMM to DRAM ratio.

The GPU code allows adjusting of the DRAM limit to accommodate GPUs with different amounts of device memory.

Additionally, creating and running new ngraph networks is relatively straight forward.

These steps are discussed in detail in the Experiment Customization section of the artifact documentation: http://arch.cs.ucdavis.edu/AutoTM/dev/customization/.

A.8 Notes

The two portions of the paper (the PMM section and the GPU section) can be performed separately. That is, the GPU portion can be run without requiring access to a Optane DC equipped server.

The code provided runs under both conditions, but requires slight configuration at setup. An interactive setup script as well as instructions are provided at the top level of the code repository. This script is executed in the root directory of the repository using

```
julia --color=yes setup.jl
```

Please feel free to file issues on the GitHub repository or contact the authors directly.

A.9 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging