Learned Memory Allocation in Heterogeneous Memory Systems

Jaewan Hong *Junsun Choi*
Computer Science, University of California, Berkeley

Abstract

Current hardware and application memory trends put immense pressure on the computer system's memory subsystem. However, with the end of Moore's Law and Dennard scaling, DRAM capacity growth slowed down significantly. To cope with this limitation, on the hardware side, the market for memory devices has diversified to a multi-layer memory topology spanning multiple orders of magnitude in cost and performance. Above from the user level, applications has increased in need to process vast data sets with low latency, high throughput.

However, memory allocation system designed to cater average cases cannot support all the demands of different memory configuration together, forcing users to optimize allocation. To solve this problem, we present a learned memory allocation scheme to automate the memory allocation, *MARL* (Memory Allocation Reinforcement Learning) for Persistent Memory(PM) and DRAM heterogeneous memory systems.

MARL outperforms current memory agnostic allocation by 3.8× and 1.2× than Intel's memory mode. More tasks could be run in parallel with MARL. Generating policy for MARL only takes 20 minutes, uncomparable to manual policy development.

1 Introduction

Computer systems are being stressed from above and from below. DRAM's scaling trend has ceased to keep up with the growing memory demands of applications. Below the computer systems, end of the Moore's Law, Dennard Scaling, and other limitations have led to the physical space limitation in DRAM scaling. Above the systems, above from the user level, applications has increased in need to process vast data sets with low latency, high throughput. Modern applications require far more capacity and performance than current systems can provide. Figure1 shows exponential growth of AI models. Memory demands grow exponentially in modern applications. Furthermore, growing low-latency services and data-intensive analytics place their data on main memory. However, homogeneous DRAM-based systems fail to meet the needs of modern applications. To cope with this limitation, the market for memory devices has diversified to a multi-layer memory topology spanning multiple orders of magnitude in cost and performance. Main memory devices are fragmented based on a trade-off between performance and capacity. Systems are now equipped with new memory technologies, such as far memory (networked memory) [11] [3], Storage Class Memory (SCM) [14], cache coherent networked memory (CXL [13], CCIX [4]), and etc.

Unlike traditional DRAM-only computer systems, future memory systems have to be fragmented. Memory systems will be comprised of various memory devices to provide larger capacity. However, equipping heterogeneous devices to compose a main memory brings several challenges. Wu et al.[20] revealed that memory agnostic allocation systems in heterogeneous memory systems (75% DRAM + 25% Persistent Memory) deteriorate the application performance down to 60%. Prior studies argue that memory objects must be efficiently allocated to minimize performance degradation.

There are a series of research proposed optimized memory allocation schemes for heterogeneous memory systems. These studies require application specific information to fully leverage the performance of memory devices. However, building a memory allocation for each memory type and application is not scalable. A number of memory technologies; 3D XPoint, STT-MRAM, PCM, ReRAM, CCIX, CXL, far memory, NVMe as memory, and etc, each with unique hardware characteristics are emerging.

Memory objects must be efficiently allocated to minimize performance degradation. For example, hot data should be placed in fast memory while cold memory could be placed in slow memory. Some access patterns

benefit from CPU. CPU prefetches sequential access and striding patterns. These memory objects can be placed in slow memory without impacting performance. However, it is impossible to make memory allocation policies to address all hardware characteristics and application properties. As mentioned above, in traditional approach where developers make a memory allocation policy, with N devices and M applications, needed $N \times M^N$ policies should be developed manually.

In the past decades, deep learning has monopolized spotlights for its promising performance, which was possible due to the recent advances in computation resources and big-data. However, the DRAM capacity growth, which is key computation resources for operation, has been gradually slowing down. In contrary, the required resources for emerging deep learning topics such as natural language processes (NLP) have been skyrocketing. As a result, today's computer industries fail to satisfy the emerging memory demands of applications.

In this paper, we focus on PM and DRAM heterogeneous memory system. Among many alternative devices, persistent memory is the only commercially available memory. MARL targets on deep learning inferences as they are expected to dominate computing demands in modern data centers. Memory demands in the deep learning workloads can be decomposed into approximately two key parts – weights and activation. The weights represent the learning parameter (typically denoted as θ) of deep learning nodes, while the activation represent the actual tensor calculation parts. We modified Pytorch[18] to hook memory allocations for weights and activations. The workloads are decomposed into a graph. Each node of a graph has one activation and one weight allocations. MARL gives allocation schemes for each node. MARL was built on top of an evolutionary graph reinforcement learning (EGRL) [15].

Our key contributions are:

- Automating memory allocation in heterogeneous memory systems with a new reinforcement learning algorithm, MARL.
- More tasks can be run in a server
- Faster runtime than baseline Intel's memory mode
- Test MARL on a real commercial server from Intel
- Open source modified pytorch to interpose memory allocation for weights and activations and MARL

MARL can be found in `https://github.com/jaewan/Mem_Alloc_RL.git`. Modified pytorch can be found in `https://github.com/jaewan/pytorch-alloc-hookup.git`. We wrote 2204 of python code for MARL and 6402 lines of C++ code to modify pytorch. We spent 300+ hours in total on this project.

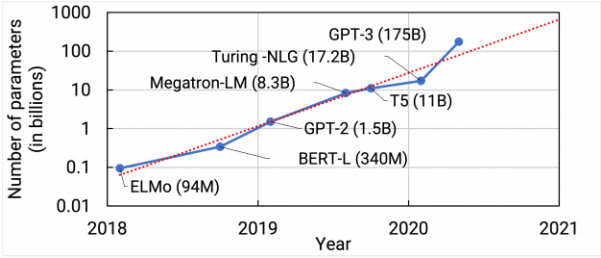


Figure 1: Exponential growth of state-of-the-art AI models figure from[5]

2 Background and Motivation

2.1 Intel Optane DC

We use Intel's OptaneDC as our persistent memory (PM) as it is the only commercially available alternative memory. PM is a solid-state high-performance byte-addressable memory. PM resides on the memory bus so that PM can have access similar to DRAM to data. Optane DC performs similar to DRAM in sequential accesses. Both read and write in sequential manner are $1-1.2\times$ slower than DRAM. However random access brings disastrous performance degradation. Random accesses slows down the performance $4\times$. Unaligned accesses are the worst. It deteriorates the performance down to $22\times$. Having these combined, small accesses from multi-threads perform the worst. There are clear improvement points in Optane DC. Avoid small unaligned accesses and favor sequential accesses.

The strongest advantage of PM is that unlike DRAM which has a limit on expanding its size in one computer system, PM can be deployed with a much larger size. Optane DC is much cheaper than DRAM but have larger capacity. It also requires less power as it does not need refresh. Thus PM is expected to revolutionize the computer systems. OptaneDC operates in two modes. Memory mode uses Optane to expand main memory capacity without persistence. It combines an Optane DIMM with a conventional DRAM DIMM on the same memory channel that serves as a direct-mapped cache for the NVDIMM. The cache block size is 64 B, and the CPU's memory controller manages the cache transparently. The CPU and operating system simply see the Optane DIMM as a larger (volatile) portion of main memory. App Direct mode provides persistence and does not use a DRAM cache. The Optane DIMM appears as a separate, persistent memory device. A file system or other management

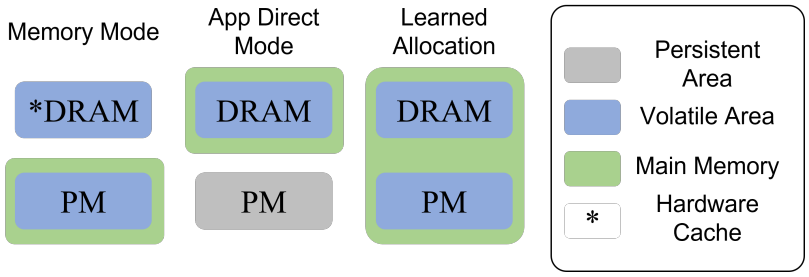


Figure 2: Available Operating modes of PM with MARL

layer provides allocation, naming, and access to persistent data.

2.2 Persistent Memory Modes

Intel provides two modes for their PM. Figure2 shows Intel's two modes Memory Mode and App direct Mode, and our proposed mode with MARL. The area in green are the components which system sees as main memory. Memory Mode uses DRAM as hardware cache and use PM as main memory only. Only PM's capacity is provided as main memory to the system. In App Direct mode, Intel mandate users map PM and manage memory allocations and life cycles. No known application runs in this mode besides file systems. File systems use app direct mode to use PM as a persistent storage. Only DRAM's capacity is shown to the system as main memory ans PM is used as storage. Our proposed system with MARL uses app direct mode in its foundation. However, MARL automates memory allocations and our fixed pytorch manages memory object lifecycle. Thus, both DRAM and PM are exposed as main memory to applications.

2.3 Related Works

In the past few years, deep reinforcement learning (DRL) has advanced rapidly, evolving from game-play benchmarks to solving massive-scale real-world optimization problems. Memory allocation is a real-world optimization problem with uncertainties, which is difficult to represent in models. Thus, we plan to exploit the model-free property of reinforcement learning (RL) to build a policy to successfully solve this problem. We will justify our selection with more detail in the next sections.

Related to adopting RL in memory problem, a recent study from Intel demonstrated that Collaborative Evolutionary Reinforcement Learning (CERL) [16] allocated memory objects more efficiently than traditional CPU schemes. The paper focuses on on-chip memory in accelerator while our problem scope resembles the problem in off-chip memory. We conjecture that RL will allocate memory objects more efficiently and adaptively than the current monolithic memory management system in OS.

More recent work from Intel[15] introduces Evolutionary Graph Reinforcement Learning (EGRL) which combines graph neural networks with CERL [16] to cope with the complexity of automating memory mapping. EGRL automates memory allocations for Intel's custom accelerator with 3 different memory types; L1 cache, L2 cache, and DRAM. Despite the large search space, the study achieved a speedup on BERT and Resnet tasks on Intel NNP-I chip using EGRL compared to using the chip's native compiler. We seek to modify EGRL to better fit on a bigger system with heterogeneous memory which would have more complex search space than a single accelerator.

To the best of our knowledge, MARL is the first study to use reinforcement learning in heterogeneous memory systems. Past research manually made memory allocation policies. X-mem[7] is the first research to propose use PM as main memory. Besides PM and DRAM heterogeneous memory systems, there are many DRAM and slow memory combination studies like remote memory. Infiniswap [11] made a hierarchy of memory like Intel's memory mode. It uses remote memory as swap space, providing local memory the only main memory. KONA[3] is the first study to integrate remote memory into main memory. However, it requires a specialized hardware with cache coherent network chips. FastSwap [1] concluded that slow memory cannot complement DRAM. RMC [2] instead built a new abstract to use slow memory. Thus in heterogeneous field, researchers focus on applications rather than studying for general memory allocation schemes for all applications. Tahoe[21] and Warpx[19] focused on HPC applications for PM and DRAM system. Autotiering [17] turned back to target general applications, but it fails to outperform Intel's memory mode.

2.4 Baseline results

Figure 3 shows Resnet-152 inference runtime over increasing number of tasks with two different memory

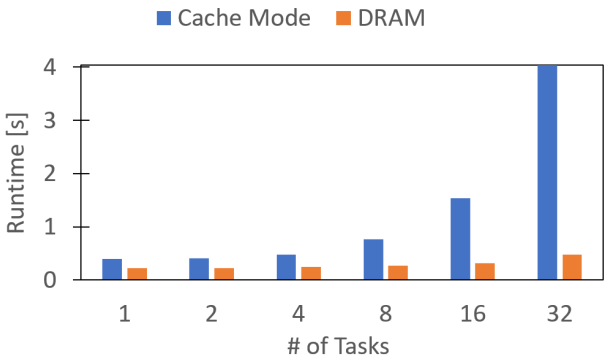


Figure 3: Resnet-152 inference runtime by number of tasks

configuration. We ran until 32 tasks to saturate CPU cores (32). The graph shows how long each inference took to run. The cache mode consists of 16GB DRAM and 128GB PM in Intel's Memory Mode using DRAM as a hardware cache. The DRAM mode in orange bars is 128GB DRAM-only configuration. The DRAM mode represents the ideal case where DRAM could satisfy the memory demand, which is unrealistic. DRAM mode does not exhibit any performance degradation. Resnet152 uses 1GB of memory. A single inference run is slower in cache mode than in DRAM only. 4 parallel tasks fill the 16GB DRAM capacity (system also uses memory). From 8 tasks, cache mode performance degradation is notable. With more working set, performance degradation gets worse. MARL aims to sit in between these two configurations.

3 Problem Formulation

As we have mentioned before, modern memory system should be equipped with heterogeneous memory types, i.e., DRAM-only systems will provide the best performance, but it cannot scale in size. There are some clear improvement points to avoid performance degradation with slow memory devices, e.g., hot data should be placed in fast memory while cold memory could be placed in slow memory. Some access patterns benefit from the CPU. CPU prefetches sequential access and striding patterns. These memory objects can be placed in slow memory without impacting performance. Besides the two general cases, there are several device specific improvement points. For example, in Intel OptaneDC (Persistent Memory, PM), small, unaligned, and random accesses severely harm the performance. Addressing these hardware characteristics will avoid performance degradation in heterogeneous memory systems.

The solution for the memory allocation problem should satisfy the following properties.

1. First, the resulting policy should not use much of its resources, which the conventional approach utilizing the dynamic programming often fails.
2. Second, the resulting policy should deal with uncertainties of the computer operating systems (OS), which make monolithic approaches fail. As it is impossible to make optimized policies for each combination of slow and fast memory, to automate memory object allocation in systems with slow and fast memory would be the best solution.

3.1 Target System

As mentioned, we saw an opportunity to enhance performance of systems using heterogeneous memory devices by optimizing storage policy. To construct a heterogeneous system, we chose the combination of Intel Optane DC and DRAM for our memory system configuration. The reason why we chose Optane DC is because it is the only commercially available one for us among next-generation memories.

Consider a memory system that consists of DRAM and Intel Optane DC. In The default mode of this system which is called the cache mode, Optane DC uses DRAM as a hardware cache, exposing Optane DC's capacity as the only main memory. While this scheme seems universally general for various types of applications, it does not address any hardware characteristics. If applications use a huge amount of memory (which is the point of using PM), using DRAM as a hardware cache does not manipulate accesses to Optane DC. Therefore, with this default memory policy, the strengths of Optane DC cannot be exploited. If DRAM and Optane DC are treated as a flat memory, the overall performance with respect to a workload will be better than the default mode.

Other than memory, CPU (2nd Gen Intel Xeon Scalable processors) only without any GPUs from linux kernel 5.1.0. is utilized. DRAM only system is equipped with 16GB DRAM and heterogeneous system is equipped with 16GB DRAM and 128GB PM. Heterogeneous memory system uses the basic Intel memory mode which caches all memory objects in DRAM first.

3.2 Target Workload

To demonstrate that our proposed solution works well with memory-intensive workloads, We selected the target workload as deep learning processes, which needs repetitive reading (execution) and writing. The state-of-the-art convolutional neural network or natural language processing networks consume a lot of memory space allocating its weight and activation with a high chance of saturating DRAM and PM. This may be from the learning process itself, or it may be from the inference stage

where the trained network is applied for real-world product. To this end, we have dissected PyTorch code below the C++ level, to hook-up native memory allocation and replace it with our RL policy. Our allocation targets are the weights and activations, where inputs have to be on the DRAM side. Weights and activations can be stored in either of the Optane DC or the DRAM to maximize inference throughput.

4 MARL Algorithm Detail

Building an RL algorithm for memory allocation was a challenging task. We have used multiple schemes introduced in RL field and integrated them together. This section explains in detail about the algorithm. If you are unfamiliar with RL, you may skil section 4. This is irrelevant to system design. However, we elaborate algorithmic detail as it is one of our key contributions.

Deep reinforcement learning (DRL) has shown promising results for having the two properties as a solution for memory allocation problem stated in section 4. After the training, the light-weighted policy does not require much resources, and the robustness of the policy with respect to uncertainties is realized through repetitive learning processes. Also, we can automate the process of finding the optimal memory allocation policy. Therefore, we decided to use reinforcement learning (RL) as our solution for memory allocation optimization.

Our goals are (1) to allocate more deep learning tasks on a machine and (2) to run faster than the cache mode.

Figure 4 shows the big picture of our solution. To train the reinforcement learning model, a reward is needed. To meet our goal, the reward is set as the time consumed to run deep learning inference in a server which consists of DRAM and Optane DC. After the RL model is trained with the reward, it returns the allocation decision to deep learning inference. This process is iterated to achieve the optimal allocation decision.

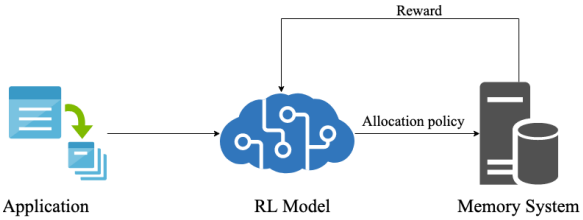


Figure 4: Big picture of RL solution

For our memory allocation problem, we will utilize MARL based on algorithm proposed in [15] , EGRL. In EGRL, the neural network computation problem was first formulated as the graph network and the graph neural network was fed to the policy and the critic network as the input. To this end, the graph neural network method suggested in [10] was employed. For the deep-learning-based workloads, a number of graph nodes are is about $\sim 10^2$, e.g, ResNet50 is composed of 53 nodes, while bigger workloads such as BERT or GPT2 have number of nodes over 200. Since the EGRL have two options (PM and DRAM) of memory allocations for each nodes, there are 2^{53} actions that the EGRL can select. Therefore, a policy exploration method become critical. Additionally, the EGRL utilizes soft-actor-critic (SAC) algorithm [12] for policy update.

4.1 Formulation into a standard Markov Decision Process

The hardware mapping problem is formulated into a standard Markov Decision Process (MDP) and we apply the DRL to solve the MDP. To this end, we introduce the settings for states, actions, and rewards in this subsection. The whole interaction process of the EGRL agent with the environment is summarized in Algorithm1.

4.1.1 States

We utilize the graph representation of the target workloads as the states, where each nodes represent the operational layers, and the edges represent the connectivity between them. In our case and EGRL, we left the edges featureless by having all the information of the workload in the outgoing nodes' features. The node feature embedding encapsulate information about input and output tensors of the operation and summary information of future layers [15]. The details of the features utilized for the node embedding can be found in Appendix A of [15].

4.1.2 Actions

The agent of the MARL receive the graph embedding as input and gives an output graph, which features are the mapping proposals for the activation and weights tensors for each nodes. The complete proposal $\mathscr{M}_\pi$ of the agent is then sent to the compiler. In fact, there are different numbers of weights and activation tensor calls, e.g., 2D convolutional layer has 3 weights call while activation tensors are called for 1 times, and 2D batch normalization layer has 3 weights calls and 5 activation calls. For the simplicity, we just dump weights and activation tensors from the same nodes into the same memory allocation. To illustrate, the tensors for weights from 2D convolutional layer from node 1 is allocated to one memory, and the tensors from activation is allocated to another, etc. For the PG part, the actions are obtained through policy $\pi_\theta(a|s)$, where the inputs s are graph representation, i.e., $s = \mathscr{G}(f)$, and the outputs are memory mapping

proposals – See Figure5

4.1.3 Reward function

Reward function of the EGRL is composed of two parts; negative reward for invalid mappings and positive reward for accelerating the speed. The action of EGRL might give the mapping that is impossible to be compiled by the compiler. For instance, the EGRL might allocate 24GB of memory to total 16GB of DRAM. To cope with this phenomenon, negative rewards are given to the agent if such invalid mappings are the outputs. Negative rewards are formulated to quantify the extent of the invalidity.

When the agent produces fully valid memory allocation, the running time of the process is then compared with those of the native compilers. The positive reward are given as scores of the agent normalized by scores of the native compilers, i.e., when the normalized score is larger than 1, the agent performs better than the native compilers.

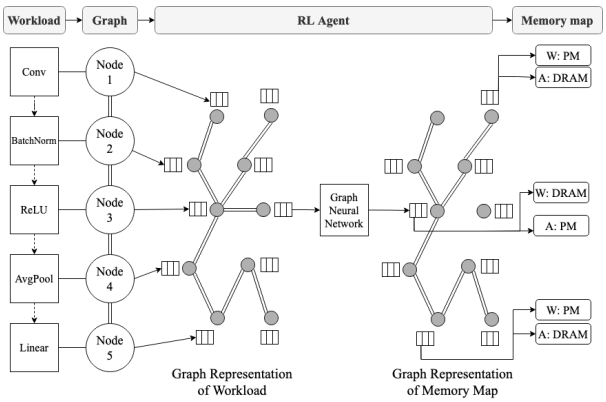


Figure 5: Optimizing memory mapping by EGRL

4.2 Training and Exploration

The EGRL algorithm builds upon the CERL framework [16]. We are going to elaborate on the MARL algorithm. The MARL algorithm builds upon the EGRL framework CERL framework [16].

4.2.1 Interaction of the agent with the environment

In MARL, GNN-based policy gradient (PG) learning and the evolutionary algorithm (EA) operate concurrently by utilizing multiprocessing. The MARL comprises of a single PG learner with a GNN architecture, and an EA population containing a mixture of GNN and stateless Boltzmann policies. Each individual in the population gives a different allocation proposal for a given input workload. These proposals are then evaluated by performing an inference run ($\mathscr{I}(\cdot)$) on the input workload and measuring the resulting running time. The tuples (s_i, a_i, r_t, s_{i+1}) generated during the evaluation process by all policy of individuals are stored in the shared replay buffer $\mathscr{R}$. For the next step, the population takes the processes of standard EA to produce a new generation of allocation candidates.

The PG learner is now updated by sampling from this replay buffer $\mathscr{R}$. This is enabled by selecting the PG learner as typical off-policy algorithms. The actor is periodically copied to the weakest population of EA as a form of information transfer. At any given time, the top-ranked policy in the EA population is chosen for deployment.

4.2.2 Boltzmann Chromosome

An additional policy representation called Boltzmann chromosome is introduced in the MARL. The Boltzmann chromosome policy is parameterized by a set of prior probabilities and associated temperature for each nodes. In contrast to GNN policy, the Boltzmann chromosome policy is significantly faster when computing the policy. Therefore, the Boltzmann chromosome is ideal for search-based EA method. For more details, please refer to [16].

4.2.3 Evolutionary Algorithm

Since there are a lot of actions available, the RL algorithm for GNN needs vast exploration. In MARL, the exploration is tackled via EA. In the EA, the population is evaluated by executing the target workload using the proposed memory allocation. A selection operator then selects a portion of the population for survival with the probability of their relative performance metric. The population undergoes probabilistical perturbation through mutation and crossover to create next generation. The top performers of selected portions are preserved as elites and are not mutated in the mutation step. The overall training step is summarized in Figure5

The weights of the PG learner network's are periodically transferred to the population pool of EA. By this process, the EA can leverage the information obtained during the gradient steps. In addition, this process stabilizes learning and enables robustness to deception.

4.3 Policy gradient as Soft Actor Critic (SAC)

For the PG, SAC [12] is adopted to cope with large multi-discrete action space. Since the policy is in discrete space, the entropy is computed directly as

$$\mathscr{H}(\pi_\theta(\cdot|s)) = \mathbb{E}_{s\sim D}\left[-\sum \pi_\theta(\cdot|s)\log\pi_\theta(\cdot|s)\right], \quad (1)$$

where D indicates the sampled transitions from the replay buffer $\mathscr{R}$. Following the EGRL setup, we also utilize the Bellman update as follows:

$$L_\phi = \frac{1}{T}\sum_i \left(y_i - Q_\phi(s_i,\tilde{a}_i)\right)^2, \\ y_i = r_i + \gamma \min_{j=1,2} Q^*_{\phi,j}(s_{i+1},a_{i+1}) + \mathscr{H}(\pi_\theta(\cdot|s_{i+1})), \tag{2}$$

where y_i are called target Q function, $Q^*_\phi = \max_{a_{i+1}} Q_\phi(s_{i+1},a_{i+1})$, and $\tilde{a}$ is given by

$$\tilde{a}_i = a_i + \varepsilon, \quad \varepsilon = clip(\varepsilon \sim N(\mu,\sigma^2), -c, c) \tag{3}$$

Here, to mitigate the drawback of exaggerated Q function, the dual-Q trick is utilized as suggested in [9]. Note that the noise ε is added to the action a. As commented in [15], noisy action makes the policy smooth and addresses overfitting to the one-hot encoded behavior output. Note also that Q function is notated as Q_ϕ to indicate that the neural network parameter for Q function is ϕ, and the actor that utilizes GNN as input is notated as π_θ for parameter θ.

Following the policy update algorithm in [16], the policy is updated using the sampled policy gradient:

$$\nabla_\theta L_\theta \sim \frac{1}{T}\sum \nabla_a \mathscr{Q}(s,a|\theta)|_{s=s_i,a=a_i} \nabla_\theta \pi(s|a)|_{s=s_i} \tag{4}$$

The overall algorithm of the MARL is summarized in Algorithm 1.

Algorithm 1 MARL algorithm

Initialize A mixed population of k policies pop_π
Initialize Replay buffer $\mathscr{R}$
Define a random number generator $r \in [0,1)$
for generation = 1,∞ **do**
  for actor $\pi \in pop_\pi$ **do**
    fitness, experiences = Rollout(π)
    $\mathscr{R} = \mathscr{R} \cup$ experiences
    Rank the population based on fitness scores
    Select the first e actors from $\pi \in pop_\pi$ as elites
    Select $k-e$ actors from $\pi \in pop_\pi$ to form set S using tournament selection with replacement
    while $S < (k-e)$ **do**
      Select $\pi_a \in e$ and $\pi_b \in S$
      if π_a and π_b are of the same encoding type **then**
        Use single-point crossover and append to S
      else
        Sample a random state s and get action $a = \pi_\theta(a|s)$
        Use a to encode the prior of the Boltzmann chromosome
      end if
    end while
    for Actor $\pi \in S$ **do**
      if $r <$ mutation probability **then**
        Mutate θ^π by adding noise $\varepsilon \sim N(0,\sigma)$
      end if
    end for
    ups = number of environment steps taken this generation
    for $ii = 1,$ups **do**
      Sample a random minibatch of T transitions (s_i,a_i,r_i,s_{i+1}) from $\mathscr{R}$
      The critic loss is calculated as (2)
      The policy gradient step is calculated as (4)
      Soft update policy as $\theta_{i+1} \leftarrow \tau\theta_i + (1-\tau)\nabla_\theta L_\theta$
      Soft update critic as $\phi_{i+1} \leftarrow \tau\phi_i + (1-\tau)\nabla_\phi L\phi$
    end for
    Copy π_θ into the population for the weakest $\pi \in pop_\pi$
  end for
end for=0

5 Implementation

5.1 Hardware-Software Interface Implementation

We modified Pytorch to interpose memory allocation for weights and activations. Further, the modified pytorch manages memory life cycle. This implementation was especially challenging as Pytorch's interface is exposed to python while the implementation is C++. We added an argument to indicate that the allocation should be interposed in torch function and modified all entailing function calls to the allocation part. Figure6 shows call graph of convolution activation. All functions involved in the call graphs are modified and this required about five thousands lines of code to cope with python and c++ interface. The allocation of each node's weights and activations are modified in this manner.

To allocate weight and activation in either of DRAM and PM selectively, we made a custom memory allocation function which decides where to allocate a memory object. We have to steal the call from each layer of our application neural network to the default memory allocation function, then route the call to our custom allocation function. This process is depicted in Figure 7.

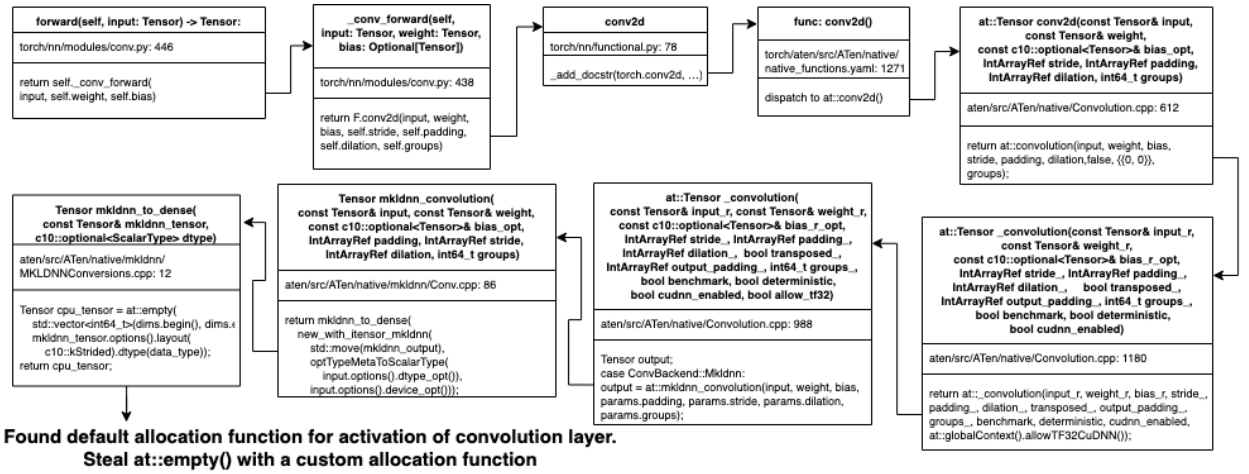


Figure 6: Call graph for activation of convolution layer

Each layer in the neural network declares its weight and activation differently, possibly using different allocation functions. Therefore, we had to track weight and activation declaration for all layers of our deep learning model for inference. After a long trace, we found that the default allocation function is at::empty() function of Pytorch. We implemented a custom version of at::empty() and stole traffic to that function like Figure 7. Our custom function uses the result from our RL model proposed in Section 5, which tells which memory to allocate the tensor. We did the same process to other layers not depicted in Figure 6, such as ReLU, BatchNorm, Linear, AveragePool, MaxPool.

We tracked the function calls from weight or activation declaration all the way to memory allocation of weight or activation. After we found which default function is used, we added a boolean type flag to the default function indicating whether the call is for allocation of weight or activation. If the call is for weight or activation, we route the call to our custom function by setting the flag True. If the flag is False, the object is allocated using the default function.

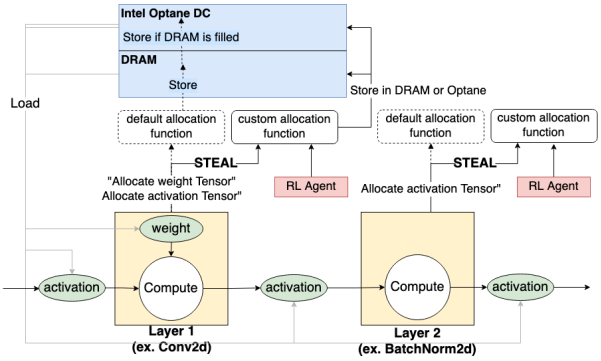


Figure 7: Stealing traffic to the default allocation function to custom allocation function

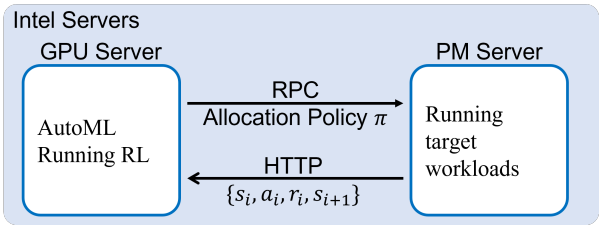


Figure 8: Experiment Server setup to run in Intel servers

5.2 Experiment Setup

Figure8 shows our experiment setup. MARL was tested on Intel's servers. MARL was deployed on a GPU server that runs AutoML. Memory mappings are passed to the PM server with RPC. RPC from AutoML invokes an inference run in the PM server. PM server returns the results back to the GPU server with HTTP protocol(for security reasons we could only use 8080 port. It was painful to work around the firewalls and make two separate nodes to work together).

6 Evaluation

We run our experiments on 16GB DRAM and 128GB PM. Samsung's DDR4 DRAM and Intel's Optane DC 3rd generation were utilized. Intel Xeon Gold CPU with 32 cores was used with one socket. Hyperthreading was turned off. We confined it to run one NUMA node to remove unnecessary memory effects from NUMA. MARL could run any deep learning inference, but with our limited time scope, we only tested with WideResnet101

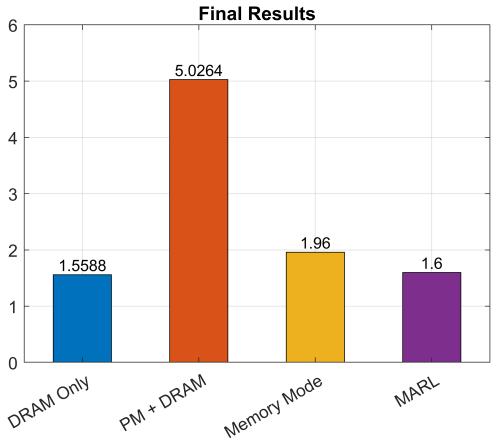


Figure 9: Runtime results for different modes.

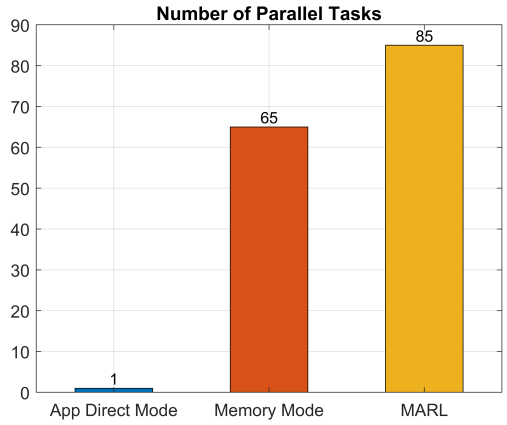


Figure 10: Number of parallel tasks run with each modes.

[22]. We have run BERT [6] and SlowFast [8],but did not run enough iterations for learning at the time of this report. Thus, we only include results from WideResnet101.

Our results show that MARL performs better than Intel's memory mode while providing more capacity to the system. More tasks can be run in parallel. An inference run faster than memory mode.

6.1 RunTime

Figure9 shows the throughput of MARL compared with those of other modes. The figure exhibits the runtime of one WideResnet101 inference run. PM + DRAM mode simulates current OS memory allocation. OS allocates memory spanning all over the memory space. We randomly allocated objects over DRAM and PM, and this random allocation ran 3 times slower than DRAM. The DRAM-only configuration performs the best. However, MARL performance is comparative to the DRAM-only performance. Interestingly, Intel advertises its Memory Mode to be the same as DRAM if the working set fits in DRAM. However, our results show that the memory mode performs slower than DRAM only system despite that the working set fits in DRAM.

6.2 Parallel Tasks

We stressed the system to the test how many parallel inference tasks a server can handle. fig:parallel tasks shows the results. In app direct mode, where system can only use 16GB main memory, only one inference could be run without out-of-memory. Since Xeon CPU requires about 14GB of system memory, only 2GB is left for tasks to use. WideResnet101 uses about 1.5GB memory, only one task could be loaded in app direct mode. Memory

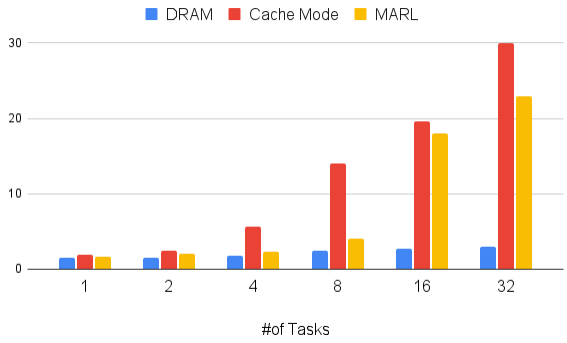


Figure 11: Mutiple WideResnet101 run of MARL compared with other modes

mode that uses DRAM as hardware cache exposes only 128GB as main memory. With DRAM, 65 tasks could be run in parallel. In comparison, MARL enables 85 tasks to run in parallel. It utilizes all 16GB + 128GB memory capacity. This result clearly shows the benefit of MARL to increase make-span of a server.

fig:Multiple WideResnet101 exhibits runtime of parallel inference runs with different modes. For DRAM only mode, we equipped our system with 128GB of DRAM(thanks to Intel to providing such servers). The graph shows how long a single tasks took to finish running. 32 tasks are run to saturate the number of cores in the server. DRAM only configuration shows almost the same runtime over parallel tasks. Intel's memory mode deteriorates dramatically from 4 tasks. This is because from 4 parallel tasks, the DRAM is saturated in memory mode. MARL performs better than memory mode in all number of tasks. Especially MARL performs near the DRAM only until 8 tasks. However, it gets drastically slow from 16 tasks. This is because same tasks running

parallel require the same allocation mapping. From 16 tasks, the same tasks scheduled on DRAM cannot be run, MARL starts to slow down. We expect this to be better with heterogeneous tasks. However, with the time limit we had, we could not run heterogeneous parallel tasks. This will be our future work.

6.3 Training Time

Figure 12 shows how MARL has trained over iteration. The higher the latency it gets lower. Iteration means one mapping tested. Figure 12 (b) shows that MARL quickly improves to 300 iterations. Figure 12(a) shows the big picture. It does not get better over more iterations. There are not much improvements from 500 iterations to 2500 iterations. This indicates that MARL does not require much iterations, just 300 iterations are enough. 300 iterations took 20 minutes. Compared to human developers making allocation policy, 20 minutes to make an allocation policy is a superb result.

7 Conclusion

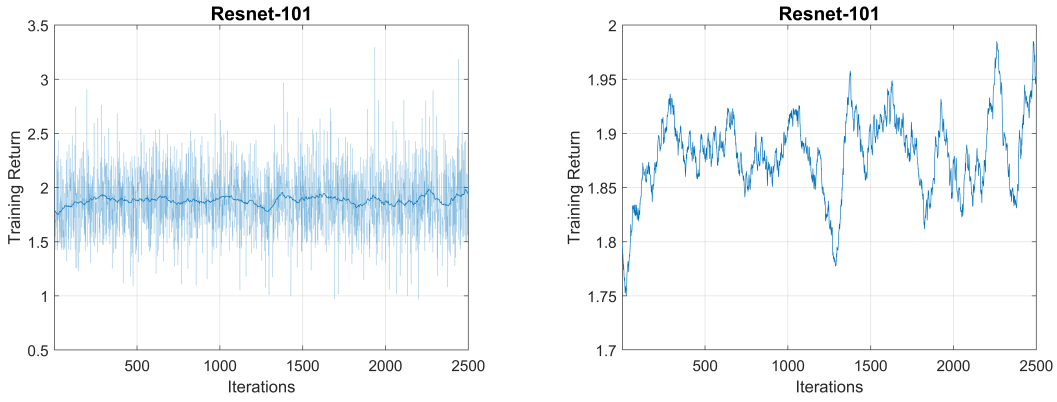
In this paper, we proposed a DRL-based approach for a memory allocation problem of deep learning tasks with heterogeneous memory. In particular, we setup heterogeneous memory system using PM and DRAM, and set the target workload as WideResNet101. The MARL algorithm is employed to solve the memory allocation problem of the deep learning tasks. As a result, we have shown that our proposed MARL approach showed $1.2\times$ improved performance compared to baseline Intel's memory mode and $3.2\times$ than ignorant operating system allocations. MARL enabled more parallel tasks to be scheduled in a single machine.

With the encouragement of positive results of MARL as memory allocation, we are planning to extend this work with more different settings. As a future work, we hope to extend this algorithm to deal with multiple target workloads, such as BERT and GPT (Tested it runs, but no results by the time of this report). We will post the results on github. Be posted if you are interested. To further improve the performance, we also hope to allocate every tensor calls of the deep learning tasks, since current algorithm regard several weights calls as one call if it is from the same node.

Our results introduced in this paper shows the eligibility of MARL in memory allocation in real systems. We will also try to adopt MARL in different heterogeneous memory systems such as local DRAM and remote DRAM.

Acknowledgments

This project was supported in part by Intel AI labs.



(a) (b)

Figure 12: Return values during the training process (a) Training curve and filtered curve (b) Filtered curve

References

[1] AMARO, E., BRANNER-AUGMON, C., LUO, Z., OUSTERHOUT, A., AGUILERA, M. K., PANDA, A., RATNASAMY, S., AND SHENKER, S. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[2] AMARO, E., LUO, Z., OUSTERHOUT, A., KRISHNAMURTHY, A., PANDA, A., RATNASAMY, S., AND SHENKER, S. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2020), HotNets '20, Association for Computing Machinery, p. 38–44.

[3] CALCIU, I., IMRAN, M. T., PUDDU, I., KASHYAP, S., MARUF, H. A., MUTLU, O., AND KOLLI, A. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 79–92.

[4] CONSORTIUM, C. CCIX Memory. `https://www.ccixconsortium.com/`.

[5] DEEPAK NARAYANAN, MOHAMMAD SHOEYBI, J. C. P. L. M. P. V. K. D. V., AND CATANZARO, B. Scaling Language Model Training to a Trillion Parameters Using Megatron. `https://developer.nvidia.com/blog/scaling-language-model-training-to-a-trillion-parameters-using-megatron/`.

[6] DEVLIN, J., CHANG, M., LEE, K., AND TOUTANOVA, K. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR abs/1810.04805* (2018).

[7] DULLOOR, S. R., ROY, A., ZHAO, Z., SUNDARAM, N., SATISH, N., SANKARAN, R., JACKSON, J., AND SCHWAN, K. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, Association for Computing Machinery.

[8] FEICHTENHOFER, C., FAN, H., MALIK, J., AND HE, K. Slowfast networks for video recognition. In *Proceedings of the IEEE/CVF international conference on computer vision* (2019), pp. 6202–6211.

[9] FUJIMOTO, S., HOOF, H., AND MEGER, D. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning* (2018), PMLR, pp. 1587–1596.

[10] GAO, H., AND JI, S. Graph u-nets. In *International Conference on Machine Learning* (2019), pp. 2083–2092.

[11] GU, J., LEE, Y., ZHANG, Y., CHOWDHURY, M., AND SHIN, K. G. Efficient memory disaggregation with infiniswap. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 649–667.

[12] HAARNOJA, T., ZHOU, A., ABBEEL, P., AND LEVINE, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *International Conference on Machine Learning (ICML)* (2018).

[13] INTEL. CXL Memory. `https://cxl.com/`.

[14] JEONG, J., HONG, J., MAENG, S., JUNG, C., AND KWON, Y. Unbounded hardware transactional memory for a hybrid dram/nvm memory system. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2020), IEEE, pp. 525–538.

[15] KHADKA, S., AFLALO, E., MARDAR, M., BEN-DAVID, A., MIRET, S., MANNOR, S., HAZAN, T., TANG, H., AND MAJUMDAR, S. Optimizing memory placement using evolutionary graph reinforcement learning. In *International Conference on Learning Representations* (2021).

[16] KHADKA, S., MAJUMDAR, S., NASSAR, T., DWIEL, Z., TUMER, E., MIRET, S., LIU, Y., AND TUMER, K. Collaborative evolutionary reinforcement learning. In *International Conference on Machine Learning* (2019), PMLR, pp. 3341–3350.

[17] KIM, J., CHOE, W., AND AHN, J. Exploring the design space of page management for multi-tiered memory systems. In *2021 {USENIX} Annual Technical Conference (ATC21)* (2021), *{USENIX}*, pp. 715–728.

[18] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., DESMAISON, A., KOPF, A., YANG, E., DEVITO, Z., RAISON, M., TEJANI, A., CHILAMKURTHY, S., STEINER, B., FANG, L., BAI, J., AND CHINTALA, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[19] REN, J., LUO, J., PENG, I., WU, K., AND LI, D. Optimizing large-scale plasma simulations on persistent memory-based heterogeneous memory with effective data placement across memory hierarchy. In *Proceedings of the ACM International Conference on Supercomputing* (New York, NY, USA, 2021), ICS '21, Association for Computing Machinery, p. 203–214.

[20] WU, K., HUANG, Y., AND LI, D. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), pp. 1–14.

[21] WU, K., REN, J., AND LI, D. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (2018), SC '18, IEEE Press.

[22] ZAGORUYKO, S., AND KOMODAKIS, N. Wide residual networks. *arXiv preprint arXiv:1605.07146* (2016).