Learned Memory Allocation in Heterogeneous Memory Systems

Jaewan Hong
Department of Electrical Engineering and Computer Science,
University of California, Berkeley
jaewan@berkeley.edu

Joohwan Seo
Department of Mechanical Engineering,
University of California, Berkeley
joohwan_seo@berkeley.edu

Ha Yun Anna Yoon
Department of Mechanical Engineering,
University of California, Berkeley
anna_yoon@berkeley.edu

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. Memory allocation system designed to cater average cases cannot support all of the demands together. We present a learned memory allocation scheme, *MARL* (Memory Allocation Reinforcement Learning) for heterogeneous memory systems. In this project, we built an reinforcement learning algorithm to automate the memory allocation automation.

Persistent memory (PM) is an alternative hardware device to complement DRAM for its slow growing capacity and power issues. PM is a promising technology to revolutionize computer systems as it provides orders of magnitude larger capacity with lower prices. However, PM exhibits peculiar characteristics. Depending on access patterns, PM performs $1 \sim 1.2\times$ to $22\times$ slower than DRAM. To leverage the full potential of PM, memory objects should be carefully placed. However, the current system mandates users to place the memory objects.

In this paper, we aim to optimize the memory allocation for deep learning tasks in the heterogeneous memory systems composed of DRAM and PM. MARL targets to run deep learning inferences, such as convolutional neural network (CNN)-based models and NLP, as they demand huge memory resource. Workloads are fed to MARL as graphs to be processed in GNN. Evolutionary algorithm produces multiple potential policies for allocation. Boltzmann chromosomes cross over to evolve into a better policy. We tested MARL on a real heterogeneous system with 128GB of PM and 16GB of DRAM. MARL enables the computer systems to have more memory capacity over state-of-the-art solutions while outperforming Intel's optimized memory mode to $1.2\times$. In DRAM-only configurations, only one inference tasks could be run while with MARL we could run 85 parallel inference tasks.

Recent advancement in deep learning inference made it plausible to run inferences on CPUs. MARL solves an inveterate memory capacity problems that exist in modern computing systems. Thus, given our results, we expect MARL to be deployed in CPU equipped inference servers. More tasks can be run on a server with MARL without manually modifying memory allocation policies.

This work was done in partnership with Intel AI Labs. They provided us a GPU server and a PM server to test MARL on a real commercial system. MARL is trained in a GPU server, and the mappings are sent to the PM server via rpc (Remote Procedure Call).

***K*eywords** Memory Allocation · Reinforcement Learning · Heterogeneous Systems

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. 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) [1] [2], Storage Class Memory (SCM) [3] [4], cache coherent networked memory (CXL [5], CCIX [6]), and etc.

Above the systems, modern applications require far more capacity and performance than current systems can provide. 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.

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.[7] 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[8] 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) [9].

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.

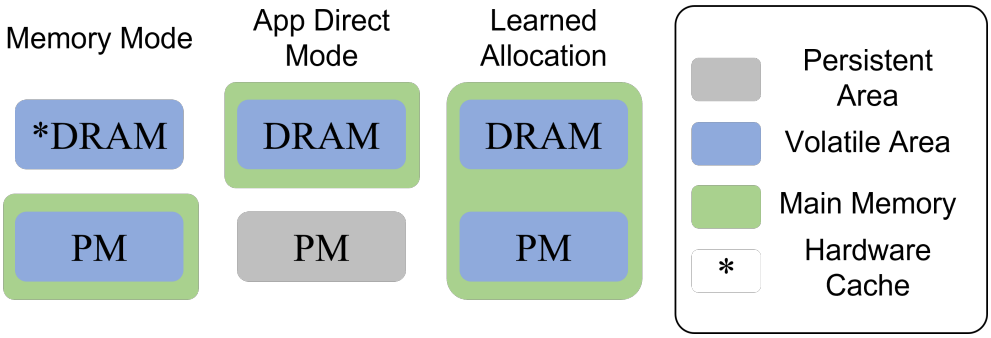


Figure 1: First the modes in the figure shows basic modes of Intel. When in memory mode, PM is utilized as the main memory and DRAM is utilized only for the hardware cache. In app direct mode, DRAM is utilized as the main memory while PM works as the storage, e.g., like hard-disks. Our proposed approach can use DRAM and PM concurrently as the main memory via proper memory allocation for specific tasks.

2 Background

2.1 Persistent Memory Modes

We use Intel's OptaneDC as our PM as it is the only commercially available alternative memory. Intel provides two modes for their PM. Figure 1 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.

2.2 Related works

Recent study from Intel demonstrated that Collaborative Evolutionary Reinforcement Learning (CERL) [10] 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. In addition, we conjecture that RL will allocate memory objects more efficiently and adaptively than the current monolithic memory management system in OS.

In Evolutionary Graph Reinforcement Learning (EGRL) algorithm proposed in [9], the neural network computation problem is first formulated as the graph network, and then the graph neural network is fed to the policy and the critic network as the input. To this end, the graph neural network method suggested in [11] is employed. GNN has indeed received much attention because the graph representation of neural networks can be useful for dealing with the connectivity of the input features. In fact, a lot of real-world data, such as social and biological networks, can be formulated and represented by the graph network [11]. Our target workload, deep-learning tasks can also be well represented by the graph network flows. Moreover, it enables the generalization over workloads of varying size and connectivity; i.e., the same algorithm can be applied to other types of network structures, such as BERT [12]. By utilizing GNN-based policy, the needs for the serialized, layer-dependent representations can be eliminated.

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[13] 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 [1] 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[2] is the first study to integrate remote memory into main memory. However, it requires a specialized hardware with cache coherent network chips. FastSwap [14] concluded that slow memory cannot complement DRAM. RMC [15] 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[16] and Warpx[17]focused on HPC applications for PM and DRAM system. Autotiering [3] turned back to target general applications, but it fails to outperform Intel's memory mode.

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 is not scalable 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. The deep reinforcement learning (DRL) has shown promising results for tackling these issues.

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.

3.1 Target System

MARL targets DRAM and PM (Intel OptaneDC) heterogeneous memory system. Currently, Intel OptaneDC is the only commercially available memory device other than DRAM. OptaneDC uses DRAM as a hardware cache, exposing Optane'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 OptaneDC. Other than memory, only CPU (2nd Gen Intel Xeon Scalable processors) 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–See Fig. 1

3.2 Target Workload

We selected the target workload as deep learning processes, which needs repetitive reading (execution) and writing. This may be from the learning process itself or 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.

The ResNet101 [18] is selected for our workload. ResNet101 is CNN-based deep layers for image processing purposes. For image process applications need to be operated multiple time repeatedly and require huge memory, it is ideal for the verification of our approach. ResNet101 is also composed of multiple layers with some feed-forward connections between the layers. This feed-forward connections make the graph representation not just serial connection, which can be utilized as the important feature for the GNN policy.

4 Solution approach - Memory Allocation Reinforcement Learning

For our memory allocation problem, we will utilize Memory allocation reinforcement learning (MARL) algorithm. In MARL, 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 [11] 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 108 nodes, ResNet101 has 311 nodes. Since the MARL have two options (PM and DRAM) of memory allocations for each nodes, there are 2^{311} actions that the MARL can select. Therefore, a policy exploration method becomes critical. Additionally, the MARL utilizes soft-actor-critic (SAC) algorithm [19] for policy update.

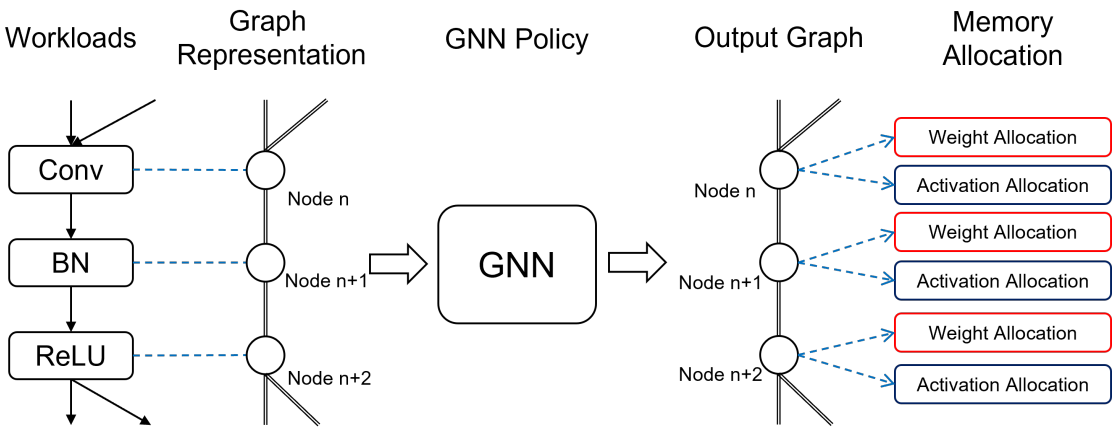


Figure 2: GNN policy (π_θ) gets an input graph representation ($\mathcal{G}(f)$) of target workload (f), and gives an output graph, where the memory allocation proposals are located in each nodes.

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 MARL agent with the environment is summarized in Algorithm 1.

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 MARL, 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 [9]. The details of the features utilized for the node embedding can be found in Appendix A of [9].

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 $\mathcal{M}_\pi$ 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., $s = \mathcal{G}(f)$, and the outputs are memory mapping proposals – See Fig. 2

4.1.3 Reward function

Reward function of the MARL is composed of two parts; negative reward for invalid mappings and positive reward for accelerating the speed. The action of MARL might give the mapping that is impossible to be compiled by the compiler. For instance, the MARL 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. To summarize, the reward is given as

$$r_i = \left(\frac{\Omega}{\Omega_{baseline}}\right)^2, \tag{1}$$

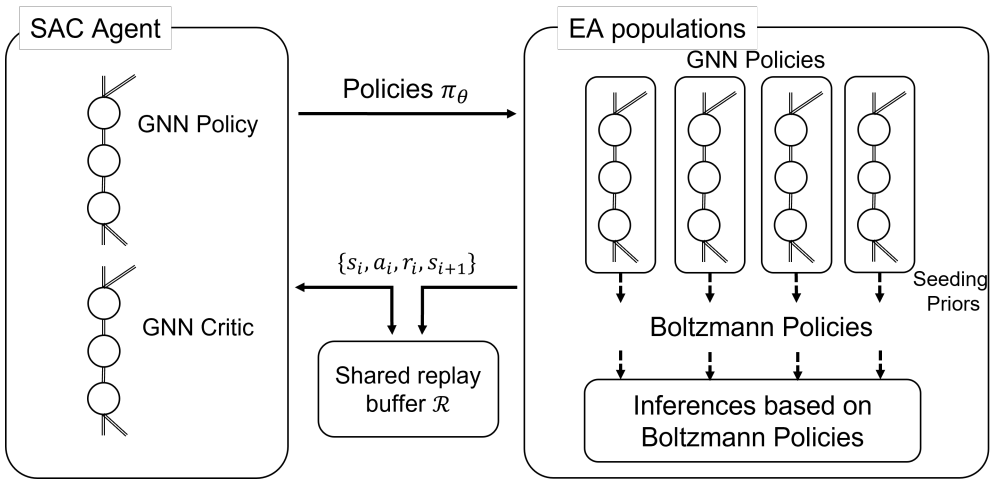


Figure 3: Interaction of GNN policy and the environment with EA algorithm is presented. SAC agent is trained with batches of state-action transition from the shared replay buffer. The EA periodically receives the SAC policies and generate Boltzmann policies. Each population run the inferences and store its results in the shared replay buffer.

where Ω indicates the latency of the training agent, and $\Omega_{baseline}$ indicates the latency of the baseline allocation rule.

4.2 Training and Exploration

We are going to elaborate on the MARL algorithm. The MARL algorithm builds upon the EGRL framework [9] CERL framework [9].

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 ($\mathcal{I}(\cdot)$) 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 $\mathcal{R}$. 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 $\mathcal{R}$. 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 [9].

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 Fig. 3

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

$$\mathcal{H}(\pi_\theta(\cdot|s)) = \mathbb{E}_{s\sim D}\left[-\sum \pi_\theta(\cdot|s)\log\pi_\theta(\cdot|s)\right], \tag{2}$$

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

$$\begin{aligned} L_\phi &= \frac{1}{T}\sum_i \left(y_i - Q_\phi(s_i, \tilde{a}_i)\right)^2, \\ y_i &= r_i + \gamma \min_{j=1,2} Q^*_{\phi,j}(s_{i+1}, a_{i+1}) + \mathcal{H}(\pi_\theta(\cdot|s_{i+1})), \end{aligned} \tag{3}$$

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

$$\tilde{a}_i = a_i + \varepsilon, \quad \varepsilon = clip(\varepsilon \sim N(\mu, \sigma^2), -c, c) \tag{4}$$

Here, to mitigate the drawback of exaggerated Q function, the dual-Q trick is utilized as suggested in [20]. Note that the noise ε is added to the action a. As commented in [9], 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 [9], the policy is updated using the sampled policy gradient:

$$\nabla_\theta L_\theta \sim \frac{1}{T}\sum \nabla_a \mathcal{Q}(s, a|\theta)|_{s=s_i, a=a_i} \nabla_\theta \pi(s|a)|_{s=s_i} \tag{5}$$

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

5 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. Figure 4 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. Each node's weights and activations are modified in this manner.

5.1 Experiment Setup

Figure 5 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 Results and Discussion

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 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 [21]. We have run BERT [22] and SlowFast [23],but did not run enough iterations for learning at the time of this report. Thus, we only include results from WideResnet101.

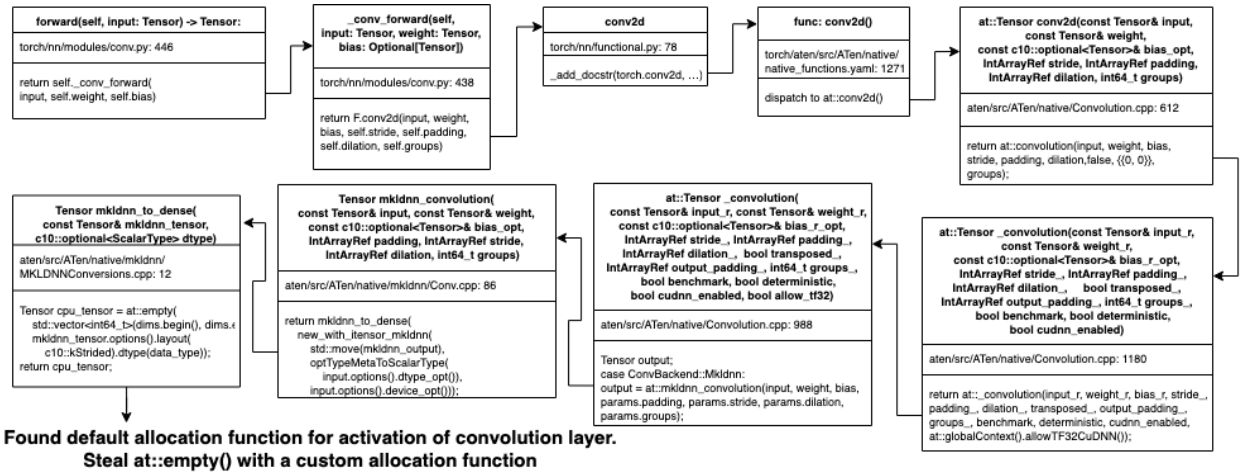


Figure 4: Call graph of convolution activation to memory allocation.

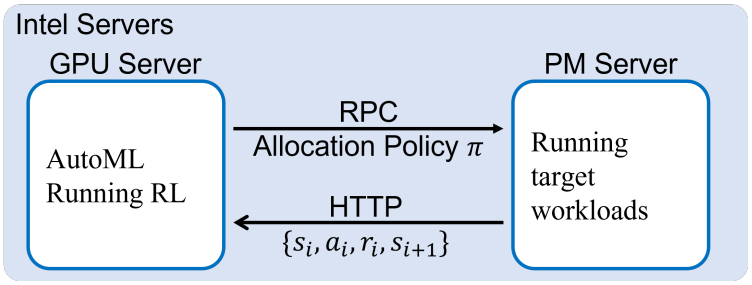


Figure 5: Our algorithm is conducted on two server environments at Intel; one is the GPU server which runs training algorithm, and the other one is the PM/DRAM server which runs our target workloads with given policy.

The training curve is plotted in Fig. 6. As soon as the training started, the return of the MARL exceeds the baseline. This result may be due to the difficulty of finding memory allocation address for the deep learning tasks–with the excessive number of possible action choices. While the baseline compiler suffers from large action spaces, the MARL efficiently find the choices using the EA-based search methods.

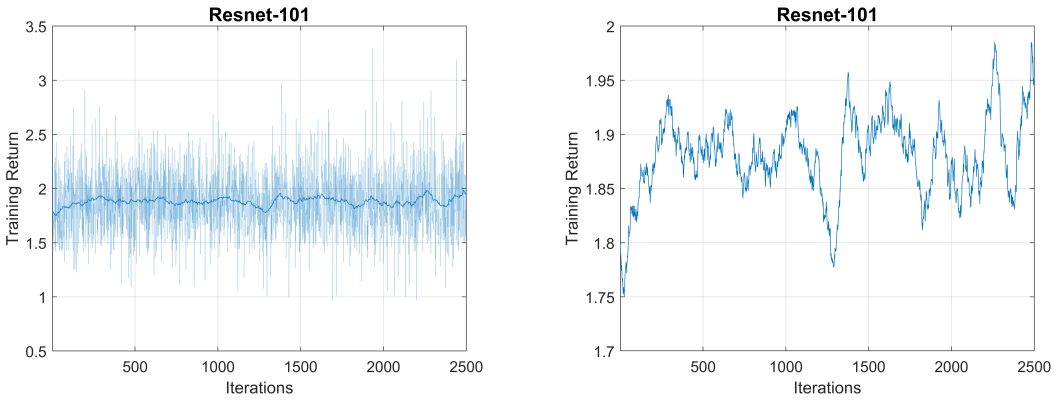
Our results show that MARL performs better than Intel's memory mode while providing more capacity to the system. More tasks can be run in parallel. An inference run faster than memory mode. :w

6.1 Parallel Tasks

We stressed the system to the test how many parallel inference tasks a server can handle. Figure 7 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 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.

6.2 Throughput

Figure 8 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



(a) (b)

Figure 6: Return values during the training process are plotted. (a) Training curve and filtered curve is plotted. (b) Filtered curve is plotted.

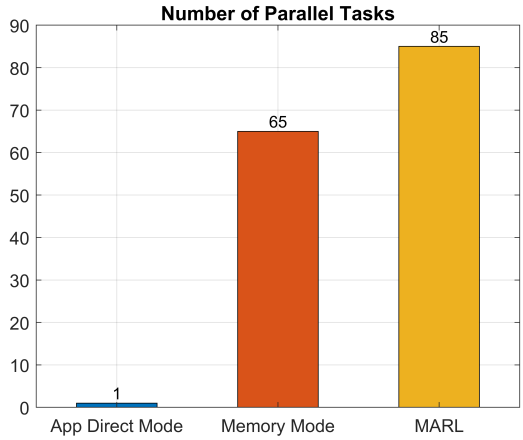


Figure 7: Number of parallel tasks run with each modes.

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.

7 Conclusion and Future work

In this report, we have proposed a DRL-based approach for a memory allocation problem of deep learning tasks with heterogeneous memory equipment. In particular, we setup heterogeneous memory system using PM and DRAM, and set the target workload as ResNet101. 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 $N\times$ improved performance compared to heuristic memory allocation.

As a future work, we hope to extend this algorithm to deal with multiple target workloads, such as BERT and GPT. 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. Unofficially, we also need to thoroughly check the whole codes to find out any possible errors, and do some hyperparameter studies that could not be conducted due to time constraints.

Acknowledgments

This project was supported in part by Intel AI labs.

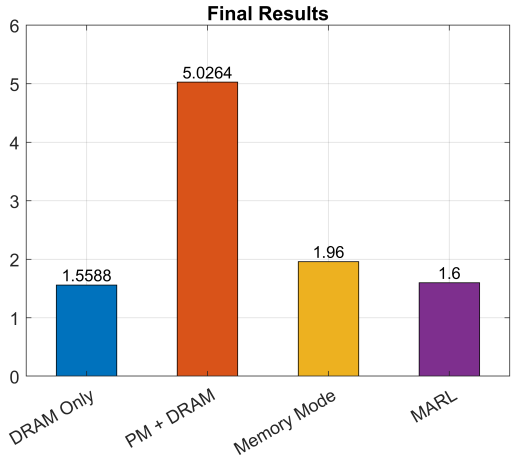


Figure 8: Runtime results for different modes.

Algorithm 1 MARL algorithm

Initialize A mixed population of k policies pop_π
Initialize Replay buffer $\mathcal{R}$
Define a random number generator $r \in [0, 1)$
for generation = 1,∞ **do**
  for actor $\pi \in pop_\pi$ **do**
    fitness, experiences = Rollout(π)
    $\mathcal{R} = \mathcal{R} \cup$ 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 $\mathcal{R}$
      The critic loss is calculated as (3)
      The policy gradient step is calculated as (5)
      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

References

[1] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 649–667, 2017.

[2] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2021.

[3] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *2021 {USENIX} Annual Technical Conference (ATC21)*, pages 715–728. {USENIX}, 2021.

[4] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. Unbounded hardware transactional memory for a hybrid dram/nvm memory system. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 525–538. IEEE, 2020.

[5] Intel. CXL Memory. `https://cxl.com/`.

[6] CCIX Consortium. CCIX Memory. `https://www.ccixconsortium.com/`.

[7] Kai Wu, Yingchao Huang, and Dong Li. 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*, pages 1–14, 2017.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[9] Shauharda Khadka, Estelle Aflalo, Mattias Marder, Avrech Ben-David, Santiago Miret, Shie Mannor, Tamir Hazan, Hanlin Tang, and Somdeb Majumdar. Optimizing memory placement using evolutionary graph reinforcement learning. *arXiv preprint arXiv:2007.07298*, 2020.

[10] Shauharda Khadka, Somdeb Majumdar, Tarek Nassar, Zach Dwiel, Evren Tumer, Santiago Miret, Yinyin Liu, and Kagan Tumer. Collaborative evolutionary reinforcement learning. In *International Conference on Machine Learning*, pages 3341–3350. PMLR, 2019.

[11] Hongyang Gao and Shuiwang Ji. Graph u-nets. In *international conference on machine learning*, pages 2083–2092. PMLR, 2019.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[13] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 38–44, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Kai Wu, Jie Ren, and Dong Li. 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*, SC '18. IEEE Press, 2018.

[17] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. Optimizing large-scale plasma simulations on persistent memory-based heterogeneous memory with effective data placement across memory hierarchy. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 203–214, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[19] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[20] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

[21] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[23] Christoph Feichtenhofer, Haoqi Fan, Jitendra Malik, and Kaiming He. Slowfast networks for video recognition. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6202–6211, 2019.