AIMM: Artificially Intelligent Memory Mapping in Near-Memory Processing System

ISCA 2022 Submission **#295** – Confidential Draft – Do NOT Distribute!!

***Abstract*—The resurgence of near-memory processing (NMP) with the advent of big data has shifted the computation paradigm from processor-centric to memory-centric computing. To meet the bandwidth and capacity demands of memory-centric computing, 3D memory has been adopted to form a scalable memory-cube network. Along with NMP and memory system development, the mapping for placing data and guiding computation in the memory-cube network has become crucial in driving the performance improvement in NMP. However, it is very challenging to design a universal optimal mapping for all applications due to the unique application behaviors and the intractable decision space.**

In this paper, we propose an artificially intelligent memory mapping scheme, AIMM, that optimizes data placement and resource utilization through page and computation remapping. Our proposed technique involves continuous evaluation and learning of the impact of mapping decisions on system performance for any workload. AIMM uses a neural network to achieve a near-optimal mapping during execution, trained using a reinforcement learning (RL) algorithm that is known to be effective for exploring a vast design space. We also provide a detailed AIMM hardware design that can be adopted as a plugin module for various NMP systems. Our experimental evaluation shows that AIMM improves the baseline NMP performance in single and multiple program scenarios up to 55% and 50%, respectively.

I. Introduction

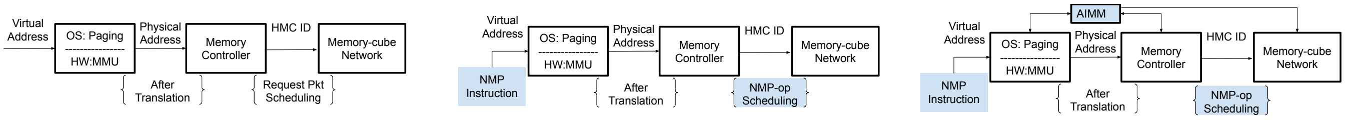
With the explosion of data, emerging applications such as machine learning and graph processing [1], [2] have driven copious data movement across the modern memory hierarchy. Due to the limited bandwidth of traditional double data rate (DDR) memory interfaces, memory wall becomes a major bottleneck for system performance [3], [4]. Consequently, 3D-stacked memory cubes such as the hybrid memory cube (HMC) [5] and high bandwidth memory (HBM) [6] were invented to provide high bandwidth that suffices the tremendous bandwidth requirement of big data applications. Although high-bandwidth stacked memory has reduced the bandwidth pressure, modern processor-centric computation fails to process large data sets efficiently due to the expensive data movement of low-reuse data. To avoid the high cost of data movement, near-memory processing (NMP) was revived to enable memory-centric computing by moving the computation close to the data in the memory [7], [8], [9], [10], [11].

Recently, memory-centric NMP systems demand even larger memory capacity to accommodate the increasing sizes of data sets and workloads. For example, the recent GPT-3 transformer model [12] has 175 billion parameters and requires at least 350 GB memory to load a model for inference and even more memory for training. As a solution, multiple memory cubes can be combined to satisfy the high demands [11]. For instance, the recently announced AMD Radeon VII and NVIDIA A100 GPU have 4 and 6 HBMs, respectively. More memory cubes are adopted in recent works to form a memory-cube network [13], [14] for further scaling up. Moreover, NMP support for memory-cube network has also been investigated in recent proposals to accelerate data-intensive applications [10].

Along with the NMP and memory system developments, memory mapping for placing data and computation in the memory-cube network has become an important design consideration for NMP system performance. Figure 1a shows an overview of the data mapping in two steps—virtual-to-physical page mapping and physical-to-DRAM address mapping. The paging system maps a virtual page to a physical page frame, then the memory controller hashes the physical address to DRAM address, which identifies the location in the DRAM. For NMP systems, the memory mapping should handle computation in addition to data, as shown in Figure 1b. Besides the data mapping, the memory controller also decides the memory cube in which the NMP operation is to be scheduled.

Prior work has focused on physical-to-DRAM address mapping to improve the memory-level parallelism [15], [16], [17]. Recently, DRAM address mapping has been investigated in NMP systems to better co-locate data in the same cubes [11]. However, adapting the mapping for the dynamic NMP application behavior is problematic due to the possibility of excessive data migration for every memory byte in order to reflect the new mappings. On the other hand, virtual-to-physical page frame mapping provides an alternative approach to adjust the data mapping during run time. Although such research has existed for processor-centric NUMA systems [18], [19], [20], [21], it has not yet been explored for memory-centric NMP systems, where computation is finer grained and tightly coupled with data in the memory system.

Furthermore, recent NMP research [7], [10] tightly couples the computation mapping with given data mapping for static offloading, which has not considered the co-exploration of the intermingling between them. This may cause computation resource under utilization in different cubes and lead to performance degradation, especially for irregular data such as graphs [22] and sparse matrices [23]. Thus, it is challenging to achieve an optimal memory mapping for NMP with another dimension of computation mapping. Moreover, the unique and dynamic application behaviors as well as the intractable decision space with NMP in the memory-cube network make it even more challenging to design a universal optimal mapping



(a) Data mapping in conventional system.

(b) Memory (data/compute) mapping in NMP system.

(c) Memory mapping in NMP system with AIMM.

Fig. 1: Memory mapping in different systems: (a) data mapping in conventional system, (b) data and compute mapping in NMP system, and (c) data and compute mapping in NMP system with AIMM.

for all types of workloads[1].

In this paper, we propose an artificially intelligent memory mapping scheme, AIMM, that optimizes data placement and resource utilization through page and computation remapping. AIMM can adapt to different applications and runtime behaviors by continuously evaluating and learning the impact of mapping decisions on system performance. It uses a neural network to learn the near-optimal mapping at any instance during execution, and is trained using RL, which is known to excel at exploring vast design space. Figure 1c depicts a system overview augmented by AIMM. In the proposed system, AIMM interacts with the paging system, the memory controller, and the memory-cube network. It continuously evaluates the NMP performance through the memory controller and makes data remapping decisions through the paging and page migration systems in the memory network. It is also consulted for computation remapping to improve NMP operation scheduling and performance. The contributions of this paper are as follows.

- An artificially intelligent memory mapping, AIMM, for adjusting the memory mapping dynamically to adapt to application behavior and to improve resource utilization in NMP systems. To the best of our knowledge, this is the first NMP work that targets mapping of both data and computation.
- A reinforcement learning formulation to explore the vast design space and to train a neural network agent for the memory mapping problem.
- A detailed hardware design and practical implementation in a plug-and-play manner to be applied in various NMP systems.
- A comprehensive set of experimental results, which show that the proposed AIMM improves the performance of state-of-the-art NMP systems in single and multi program scenarios up to 55% and 50% respectively.

The rest of the paper is organized as follows. In §II, we discuss the background and related work. Next, we state the problem and formulate the proposed approach as a reinforcement learning problem in §III. In §IV, we propose a practical hardware design for AIMM followed by the evaluation methodology in §V. Then, we present and analyze the results in §VI. Finally, we brief a case study in §VII and conclude this work in §VIII.

II. Background and Related Work

In this section, we discuss the technological advancements pertinent to NMP, followed by the computation and data mapping policies in the existing techniques. We also introduce the reinforcement learning (RL) background for data/computation remapping.

A. Logic and DRAM 3D/2.5D integration

HMC [5] and HBM [6] with logic and DRAM dies in the same silicon have enabled high bandwidth memory-centric design and near-memory processing. HMC uses packet-switching protocol to simplify capacity scaling by chaining multiple cubes as a memory-cube network [10]. On the other hand, HBM is tightly coupled with the processor using direct connections and provides the flexibility of memory controller design for command scheduling. In this work, we evaluate AIMM on a system with large chained HMC network, and also conduct a case study in §VII on capacity scaling using recently proposed HBM-PIM [24] in context of NMP operation and data (re)mapping.

B. Near-Memory Processing

The memory-centric design paradigm can be broadly classified as (1) processing using memory (PUM) [25], [26], [27], [28] and (2) near-memory processing (NMP) [29], [30]. PUM typically augments memory circuit for analog computation in the memory [25], [26], [27], [28] or cache [31]. We focus on NMP in this work, where processing units (PUs) are built on the logic layer of HMC [7], [8], [10], separate from the DRAM layers. In NMP, operations are offloaded from the processor to the memory in various granularity, ranging from operation-level [9], [7], to cache block [10], page [32], and kernel level [8], [11], where NMP system complexity varies accordingly. As operation-level offloading is the most flexible and practically feasible [24] in an augmented memory system, we choose those NMP systems as our primary target for data and computation mapping optimization.

C. Computation and Data Mapping in NMP

Existing NMP proposals such as PEI [7] and Active-Routing (AR) [10] focus more on offloading and hardware design while leaving computation and data mapping unexplored. For computation mapping, the location decisions and operation offloading schemes vary across different techniques. For instance, AR sends the operations as `<&dest += &src1 OP &src2>`, where each of the operands carry their physical addresses. On the other hand, PEI fetches one of the source data from the cache and sends the request along with that

[1] We conservatively calculate the decision space by taking the minimum number of pages touched (n pages) in any epoch in our set of applications, and mapping them to any of the m cubes, which constitutes = m^n, where $MIN(n) = 12$ and $m = 16$ in our system $\approx 10^{14}$ with the average value of n in an epoch ranges from $\approx 10s - 1000s$.

data. The computation point for each of the operations is decided following fixed policies, which either maps to closest to both the sources, or at the destination. The result of the NMP operation is returned to the processor as a response. We evaluate AIMM on two variations (created based on their computation points) of AR and PEI described in §V.

For data mapping, AR and PEI rely completely on the OS for data allocation and are oblivious to the data location. They can reduce data movement between the processor and the main memory significantly, but do not address the cost of individual operation with operands being spread across the memory network. To address this issue, we employ modified TOM [11], as a heuristic-based solution, which originally facilitates data mapping for GPU NMP systems. TOM's mapping decision is applied to a host of pages and it fails to adapt to fast changing application behaviors such as graph processing. Owing to the observation that NMP applications exhibit a fine-grained relationship between the data and computation location, more adaptive and robust solution is required.

D. Data Locality in NUMA and Multi-Program Workloads

System supports for data and computation mapping for NMP is still in its infancy. In NUMA systems, heuristic-based approaches were proposed to co-locate data and computation for better locality, including different aspects of data allocation optimization and reallocation using migration [33], [34], [35], [36], [37], [38], [39], [40], [41], [42]. Multi-program workloads make the problem even more complex due to inter-process interference. In traditional multi-program workload systems, a thread-private allocator like HOARD [33] can be used to improve the data locality by ensuring the proximity of memory accesses in a thread through bulk allocation and per-thread free lists. In this work, we explore AIMM on top of HOARD in multi-program scenarios, intending to reduce inter-process interference for NMP-ops in addition to data and computation location optimizations.

E. Migration and TLB shootdown

Apart from migration in CPU-based NUMA [42], [43], page migration is also proven to be useful in multi-GPU shared memory systems. The "first-touch" approach [44], [45], [46] can localize memory allocation to the first thread or the first GPU that accesses the memory, but does not guarantee the locality in the future computations. Baruah et al. [47] proposed to select pages and migrate them at runtime to the highly predicted GPU. In AIMM, we co-explore both data movement and computation scheduling to improve resource utilization.

Page migration needs to update the page table, typically accompanied by a TLB shootdown for maintaining TLB coherence [48]. Commercial x86-64 processors do not support automatic invalidation of stale TLB entries on a page table update. Instead, the OS is responsible for TLB coherence maintenance via the expensive TLB shootdown process, which is an active research area [49], [50], [51]. In this paper, we use hardware-assisted TLB shootdown by employing a lazy invalidation approach as described in §IV-C.

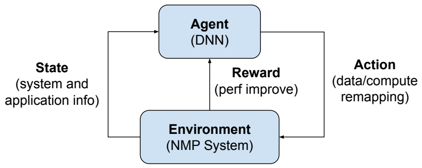


Fig. 2: A typical reinforcement learning framework.

F. Data/Computation (Re)Mapping using RL

Reinforcement Learning (RL) is a machine learning method where an agent explores actions in an environment to maximize the accumulative rewards [52]. In this paper, RL is used to design an agent that explores data and computation remapping decisions (actions) in the NMP system (environment) to maximize the performance (rewards). Figure 2 shows a typical setting of RL, where an agent interacts with the environment $\mathcal{E}$ over a number of discrete time steps. At each time step t, the agent observes an environment state $s_t \in \mathcal{S}$ *(state space)*, and selects an action $a_t \in \mathcal{A}$ *(action space)* to change the environment according to its policy π, which is a mapping from s_t to a_t. In return, it receives a reward r_t and a new state s_{t+1} and continues the same process. The accumulative rewards after time step t can be calculated as

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{1}$$

where $\gamma \in (0,1]$ is the discount factor and T is the time step when the process terminates. The state-action value function $Q_\pi(s,a)$ under policy π can be expressed as

$$Q_\pi(s,a) = \mathbb{E}_\pi[R_t|s_t = s, a_t = a] \tag{2}$$

the objective of the agent is to maximize the expected reward.

In this paper, we use Q-learning [53] that selects the optimal action a^* that obtains the highest state-action value (Q value) $Q^*_\pi(s,a^*) = \max\ \mathbb{E}_\pi[R_t|s_t = s, a_t = a^*]$ following the *Bellman equation* [52]. The Q value is typically estimated by a function approximator. In this work, we use a deep neural network with parameters θ as the function approximator to predict the optimal Q value $Q(s,a;\theta) \approx Q^*(s,a)$. Given the predicted value $Q(s_t,a_t;\theta)$, the neural network is trained by minimizing the squared loss:

$$L(\theta) = (y - Q(s_t,a_t;\theta))^2, \tag{3}$$

where $y = r_t(s_t,a_t) + \gamma\ \max_{a'}\ Q(s_{t+1},a';\theta|s_t,a_t)$ is the target that represents the highest accumulative rewards of the next state s_{t+1} if action a_t is chosen for state s_t.

RL has been widely applied to network-on-chip (NoC) for power management [54], loop placement for routerless NoC [55], arbitration policy [56], and reliability [57]. It has also been used for designing memory systems, such as prefetching [58] and memory controller [59]. Additionally, it has been applied to DNN compilation and mapping optimization [60], [61], [62]. In this work, we use RL for co-exploration of data and computation mapping in NMP systems.

III. Proposed Approach

In this section, we introduce our proposed approach by first presenting the overview and problem formulation. Then we describe the representations of state, action and reward function for AIMM NMP memory mapping, followed by the DNN architecture and training of the RL-based agent.

A. Overview and Problem Formulation

AIMM continuously evaluates the data mapping done by the OS and computes cube decisions made by the memory controllers. Meanwhile, it provides suggestions to change not only the decision if necessary, but also the degree of the change (such as migration distance) in some cases. RL agent also learns the remapping and migration cost from the system feedback for each of its actions, and decides accordingly in the future (shown in Figure 3 ❶). The problem formulation is a challenging process due to the problem complexity and the lack of straightforward methodical ways. We explore two representations in this research: (1) Dataflow graph representation of accessed pages of an application; (2) Access history of each page along with the associated events (page miss, row-buffer hits, and vault accesses, etc.). The first one involves costly hardware resources for storing the dataflow graph and expensive Node2Vec embedding [63] for processing the states. So we chose the second one that is more cost-effective. We detail the representation of states, actions, and reward function of our formulation as follows.

1) State Representation: Based on domain knowledge, the state representation (ten parameters) consists of system parameters (four) and page access history (six), as summarized in Table I. The RL agent takes this state information to improve the system performance through dynamic page-frame mapping. System information helps the RL agent to be aware of the system, while page information helps to focus on each page while taking an action. After collecting the required information of variable vector lengths, each of them is flattened into single vector of size 128×8 Bytes (128 64-bit entries). This leads to a vast space with $2^{128\times64} = 2^{8192}$ states in total in our system, making the problem very challenging.

System Information		Page Information	
Parameter Description	Vector Length	Parameter Description	Vector Length
NMP-op tab cccupancy	n cubes	Page access rate	1
Row-buffer hit rate	n cubes	Migration per access	1
MC queue occupancy	m MCs	Hop count, Pkt lat	History length
Global actions	History length	Migration lat, Page action	History length

TABLE I: The state representation of AIMM.

2) Action Representation: The actions are categorized into (1) computation migration, (2) page migration, and (3) training interval, as shown in Table II. In addition, the default action (no change) allows the RL agent to agree with the conventional system. "Near" and "Far" map to a randomly chosen neighboring HMC and the diagonally opposite remote HMC, respectively. Source co-location under page migration allows the system to co-locate NMP-op pages as a special case of page migration. The training interval regulates the epoch length in a range, discussed in §V.

Default	Computation Migration		Page Migration			Training Interval	
No change	Near	Far	Near	Far	Co-locate sources	Increase	Decrease

TABLE II: List of actions sorted categorically.

3) Reward Function: As one of the most important components of the RL system, the reward function draws significant research attention [64], [65], [66] and demands a rigorous process (left for future work). We manually explore several reward functions, including use of magnitude of performance as the reward, which shows a slow or sometimes fluctuating learning curve. Hence, we digitize our reward function that returns a unit of positive (+1) and negative (-1) reward for performance improvement or degradation, respectively. Otherwise, a zero reward is returned. We have explored using the communication hop count as a performance metric, but it leads to a local minimum that does not reflect the performance goal. We empirically found that operations per cycle as a direct reflection of performance can achieve a robust learning process.

B. RL Agent

We use the off-policy, value-based deep Q-learning [53] algorithm for the proposed RL-based AIMM. For the state-action value estimation, we use a dueling network as a function approximator. The DNN model in the agent is a simple stack of fully connected layers, as shown in Figure 3 ❸. The agent takes the state and predicts the state-action value for each action. Then we use an ε-greedy Q-learning algorithm [52] to trade off the exploitation and exploration during the search and learning process. The algorithm selects an action randomly with probability ε to explore the decision space, and choose the action with the highest value with probability $1-\varepsilon$ to exploit the knowledge learned by the agent. To train the DNN, we leverage *experience replay* [53] by keeping the past experiences in the replay buffer and randomly draw the samples for training. Therefore, the learning and search process is more robust by consolidating the past experiences into the training process.

IV. Hardware Implementation

In this section, we detail hardware modules for implementing AIMM. As shown in Fig. 3 ❷, a CMP is connected to an HMC network through a set of memory controllers (MC) on the CMP side. The MCs on the CMP side (i) map the physical address to a specific HMC, (ii) generate HMC packets, and (iii) communicate with the RL agent. On the HMC side, each NMP-op is offloaded to an HMC as computation location based on the NMP technique, and corresponding memory operations are scheduled by the vault controllers as shown in Figure 3 ❹.

A. Information Orchestration

Table I shows that the states are constituted using both system and page information, which are periodically collected at runtime. In MCs, the NMP-op occupancy is tracked for each HMC nodes stored in an array of registers, along with the MC queue occupancy information. Global actions are recorded by the RL agent itself. Row-buffer hit rate, however, is collected on the HMC node for each access and communicated to the MCs

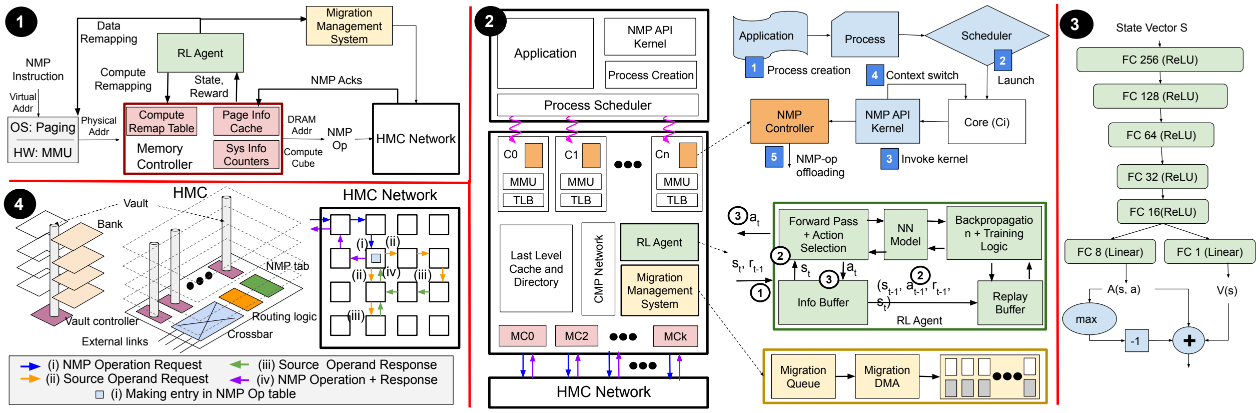


Fig. 3: AIMM reinforcement learning framework and system architecture: (1) overview of AIMM memory mapping for data and computation in NMP systems, (2) AIMM architecture, and (3) Dueling network for RL-based agent. (4) HMC network, where we show an example of basic NMP operation in the HMC network (where (i) NMP-op request enters through the corner HMC and makes entry in the NMP-op table, (ii) sends request packets, (iii) gets responses and (iv) computes in the NMP-op at the entry location), along with HMC dissection, featuring major HMC components.

on the CMP side through the HMC network, in regular interval. Page specific information is stored in a fully associative cache like structure, using page number for identifying an entry. Each entry stores the frequency of page accesses (computed and stored on each access), frequency of migration for the selected page (incremented on each migrations), and history of hop count. Packet latency, migration latency, and actions taken for the selected page are also recorded. The history entries are temporarily stored in the HMC node using history-length number of registers and spilled into the MCs whenever the history registers get filled up or collection period/window ends. The cache uses the least frequently used replacement policy. The state is constructed by combining all the information, and the reward is computed by comparing the current performance parameter value with the last computed value.

B. RL Agent Implementation

We propose to use an accelerator (as shown in Figure 3) for deep Q-learning technique following such accelerators that have been proposed in literature [67], [68]. The agent, running on the accelerator, pulls information from each memory controller. The incoming information, including the new state s_t and a reward r_{t-1} for the last state-action (s_{t-1}, a_{t-1}), are stored in the information buffer (①). The incoming information and the previous state and action in the information buffer form a sample $(s_{t-1}, a_{t-1}, r_{t-1}, s_t)$, which is stored in the replay buffer (②). Meanwhile, the agent infers an action a_t on the input state s_t (②). The generated action a_t is stored back to the information buffer and transmitted back to the MCs which then perform the appropriate operations (③). Upon the training time, the agent draws a set of samples from the replay buffer for training and applies a back propagation algorithm to update the DNN model.

C. Page and Computation Remapping

Page migration is managed by migration management unit and employed in the HMC network using migration DMA as shown in Figure 3 ❷. The TLB shootdown is customized to offset its latency overhead incurred when handled by the OS. Compute migration is implemented in the page granularity, meaning once the computation is migrated to a location, all the computation involving that page as destination (`dest`) will be done in that location.

1) Page Remapping: It involves OS for page table update and the memory network for page migration to reflect a page remapping, where the virtual page is mapped to a new physical frame belonging to a memory cube suggested by the agent. We provide *blocking* and *non-blocking* modes for pages with *read-write* and *read-only* permissions, respectively. For blocking migration, the page is locked during migration and no access is allowed in order to maintain coherence, which also experiences TLB shootdown overhead, implemented using constant latency. For non-blocking migration, the old page frame can be accessed during the migration to reduce performance overhead. In addition, the TLB is extended with an extra physical address field, which is used to hold the new frame number during migration [69]. When a page migration is requested by the data remapping decision from the agent, the page number and the new host cube are put into the migration queue of the migration management system. When the migration DMA can process a new request, the OS is consulted to provide a frame belonging to the new host cube and broadcast a message to update the extended slot for the new physical frame location in the TLBs. Then DMA starts generating migration requests that request data from the old frame and transfer it to the new frame in the new host cube. Once migration finishes, a migration acknowledgement is sent from the new host to the migration management system, which then reports the migration latency to the memory controller. Meanwhile, an interrupt is raised to invoke the OS for a page table update and notify the TLBs to use the new physical frame. In case of blocking migration, the page is unlocked for accessing, whereas for non-blocking migration, the old frame is put back to the free frame pool when the outstanding accesses finish.

2) Computation Remapping: Computation remapping decouples computation location and the data location for balancing load and improving throughput of the NMP memory network. Computation cube of an NMP-op is determined by the NMP-op

Hardware	Configurations
Chip Multiprocessor (CMP)	16 core, Cahce (2MB/each core), MSHR (16 entries)
Memory Controller (MC)	4, one at each CMP corners, Page Info Cache (128 entries)
Memory Management Unit (MMU)	4-level page table
Migration Management System (MMS)	Migration Quaue (128 entries), DMA (Rx,Tx buffers 128 entries)
Memory Cube	4GB, 32 vaults, 8 banks/vault, Crossbar
Memory Cube Network (MCN)	$4{\times}4$/ 8×8 mesh, 3 stage router, 128 bit link bandwidth
NMP-Op table	512 entries
Hyper-parameters	reply-buffer (1000 entries), minibatch (128), learning rate (0.005)

TABLE III: Hardware configurations.

scheduler based on the data address. The computation cube is then embedded in the offloaded NMP-op request packet. A compute remap table is used to remap the computation to a different cube suggested by the agent. When a compute remapping decision related to a page is given by the agent, the page number and the suggestion are stored in the compute remap table. Upon scheduling an NMP-op, the NMP-op scheduler consults the compute remap table. If the related page of the NMP-op has an entry in the table, the computation cube is decided based on the agent suggestion recorded in the entry. Otherwise, the default scheduling is used.

V. Evaluation Methodology

In this section, we describe the simulation framework and methodology that implements and evaluates several NMP techniques and mapping schemes, respectively, followed by the workloads and analysis of their characteristics.

A. *Simulation Framework*

We develop a fast and accurate simulation framework (AIMM-sim) to model RL agent and system architecture. The RL agent is functionally modeled using keras-rl [70] and built upon gym [71]. Since we propose to use a hardware accelerator for RL implementation (as discussed in §IV-B), the timing aspect of the hardware accelerator is extracted using the MAESTRO inference model [72], considering training time as 2× of the inference time. The simulation framework takes an event-driven approach to model a cycle-level system architecture with three group of components, namely, (1) Front-end (2) Chip-Multi Processor (CMP) and associated components, and (3) HMC Model and HMC network. The RL agent and the system architecture model are seamlessly integrated in order to achieve high simulation speed. The hardware configurations used in our evaluation are summarized in Table III.

1) Front-end: The front-end of AIMM-sim supports writing micro-kernels using a simple programming interface (equipped with overloaded memory allocation API, memory access API, etc). The trace manger facilitates real application traces and derives NMP operations. The processes that are registered during the initialization phase are launched on their allocated cores at their specified launching time by the process scheduler.

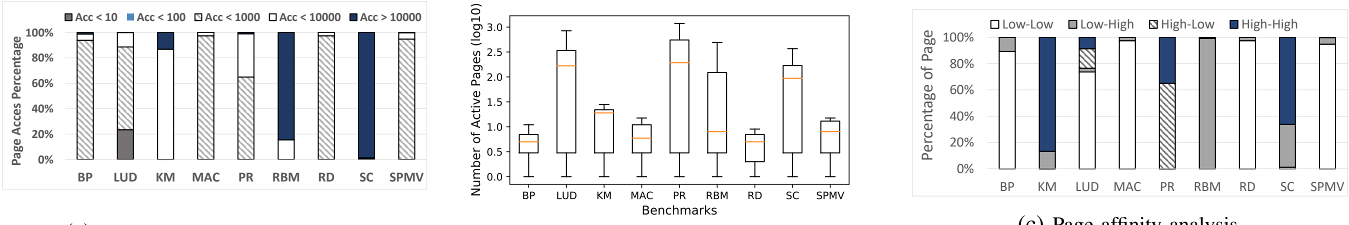
2) CMP and Associated Components: The event-driven implementation of CMP consists of a set of in-order cores, four Memory Controllers (MC), MMU units, and an NMP controller connected through a fixed latency high-bandwidth crossbar (to achieve high simulation speed). In a 64GB unified memory network formed using sixteen 4GB HMCs, MCs are responsible for determining the host cube of a given 36-bit physical address with 4 bits, and the remaining 32 bits are used to address (4GB) in each individual HMC. In addition, MCs create and send packets to respective HMCs, and also host page info caches to record page related information. The CMP, bus and the MCs have their respective queues, which backpressure NMP operation offloading; when they are full, forcing processors to stall fetching. The NMP controller helps create and offload the NMP operations to the HMC after the virtual-to-physical address translation, facilitated by the MMU unit equipped with a functional 4-level page table. The MMU also helps keep track of the migration candidates, and invokes DMA to initiate physical data migration through the memory network. Page faults are realized by employing process stall for a fixed number of cycles.

3) HMC Model and HMC network: The cycle accurate HMC network consists of event-driven HMC cubes [73] connected through the NI with the 3-stage pipelined router logic [74], and high bandwidth external SERDES links [75], [76]. The protocol deadlock is avoided using separate virtual channels (VCs) and network deadlock is avoided by using static XY routing. We aim for high-speed network simulation with simple round-robin allocation for the VCs and switch. The HMC reflects two different latency situations (row buffer hit and row buffer miss). Each of the banks has a row-buffer and each of the vaults has 2 banks connected in each layer (total 4 layers), constituting a total of 8 banks per vault. Parallel vault (32) accesses are facilitated by their respective vault controllers. NMP operations, are stored in an NMP-op table in each HMC, where each entry in the table connects to a simple Processing Unit (PU). The PUs receive operands either from the vault of the local cube, or from the remote cube through the crossbar that connects all the vaults and the NMP-op table with the external links. We are prepared to make the framework public after the decision is made.

B. *Simulation Methodology*

AIMM-sim gives a parallel simulation view by running the hardware system and the RL agent in alternating epochs (100 to 250 cycles), where the RL agent recommends on the basis of system states in the last epoch to take action in the current epoch. The majority of the simulations are trace based, where the traces are collected by executing applications with data set-sizes ranging from 80 MB to 0.5 GB and annotating NMP-friendly regions of interest that were identified in previous works [7], [10]. The traces of an application form an episode for the application. AIMM evaluates both single-program and multi-program workloads, for several episodes (5 to 10), clearing the hardware system after each episode. The combinations of multi-program workloads are decided based on the workload analysis (§V-C). We evaluate the AIMM performance on the following techniques.

We create two variations of Active-Routing (AR) technique, based on the NMP-op computation point and named as (1) Basic NMP (BNMP) and (2) Load Balance NMP (LDB). The BNMP computation point is the HMC that contains the `dest` page of the requested NMP-op `<&dest += &src1 OP`



(a) Page-access based classifications.

(b) Page touched distributions.

(c) Page affinity analysis.

Fig. 4: Workload analysis graphs. (a) Classification of pages based on their access volume, (b) Page touched distributions representing number of pages touched in each epoch of 10000 cycles for the whole application execution, (c) Page affinity showing the interrelation among the pages in an application.

`&src2>`. The LDB computation point is the `src1` operand location, owing to the observation that `src` operands span across more pages than `dest`, and hence are used for default load balancing. We use PEI unchanged as they send one operand data with the NMP-op request itself; the computation point is always the other operand location.

The TOM heuristic is adopted on top of the static computation point logic to improve performance through better resource management. It derives the node-id using a few bits from the physical address by using hash functions, decided after profiling for an epoch (10000 cycles) and applied in the following epoch. Multi-program scenarios make use of HOARD for localizing process data through page frame allocation using process specific free-frame list, in order to avoid inter-process interference. Unless otherwise mentioned BNMP is considered as our common baseline technique for result normalization and NMP operations are marked as non-cacheable [10].

C. *Workload analysis*

We primarily target the long running applications with large memory residency, which repeatedly use their kernels to process and compute on huge numbers of inputs. The machine learning kernels are a natural fit for that as they are widely used to process humongous amounts of data flowing in social media websites, search engines, autonomous driving, online shopping outlets, etc. These kernels are also used in a wide variety of applications such as graph analytic and scientific applications. Since these are well known kernels, we describe them briefly in Table IV. To capture the page-frame mapping related traits of the application, we characterize the workloads in terms of (1) page usage as an indicator of page life-time, (2) number of pages actively used in an epoch as an indicator of space requirement in the page information cache, and (3) inter-relation among the pages (page affinity) accessed to compute NMP operations in order to analyze the difficulty level for optimizing their access latency.

Benchmarks	Description	Input Data Size
Backprop (BP) [77]	Training feedforward neural networks.	2097152 hidden units
LUD [77]	Product of a lower and upper triangular matrix.	4096 matrix dimension
Kmeans (KM) [78]	Iterative algorithm, partitions the data-set into K pre-defined distinct non-overlapping clusters.	mfeat-zer [79]
MAC	Multiply-and-accumulate over two sequential vectors.	2×6400K dimension
Pagerank (PR) [2]	Rough estimation of the importance of the webpages.	web-Google graph [80]
RBM [1]	Variant of Boltzmann machines [81].	Variant of Netflix database [82]
Reduce (RD)	Sum reduction over a sequential vector.	6400K dimension.
SC [83]	Assigns each point of a stream to its nearest center.	10 20 128 16384 16384 1000
SPMV [78]	Solve sparse linear systems.	4096×4096 matrix

TABLE IV: List of benchmarks and corresponding input data sizes.

1) Page Access Classification: The page usage is an indicator of its scope for learning the access pattern and improving performance after data or computation remapping at runtime. If the number of accesses to pages are very low, the scope for improvement using remapping technique narrows down to a great extent. In Figure 4a, we show that for most of our application kernels, the majority of the pages are moderate to heavily used over the whole application execution, which offers a substantial scope for AIMM.

2) Page Touched Distribution: Figure 4b shows the number of pages touched in an epoch of 10000 cycles. We observe low page reuse across epochs, leading to marginal benefits from using caches, which gets further amortized by high data movement cost. Hence, NMP offloading can save latency and bandwidth usage for these workloads. We can clearly identify two classes of applications. (1) High number of touched pages, *LUD, PR, RBM, SC*, and (2) Low or moderate number of touched pages, *BP, KM, MAC, RD, SPMV* in the epochs. As opposed to the medium to large working set (80MB to 0.5GB) of these applications [10], this study gives us an indication of the amount of page information ideally needed to be stored in the page information cache in an epoch for the training of the RL agent.

3) Affinity Analysis: In data mining, affinity analysis uncovers the meaningful correlations between different entities according to their co-occurrence in a data set. In our context, we use and define the affinity analysis to reveal the relationship among different pages if they are being accessed for computing the same NMP operation. We track two distinct yet interlinked qualities of page access pattern, (1) the number of pages related with a particular page as the radix for that page, which is similar to radix of the node in a graph, (2) the number of times each pair of connected nodes are accessed as part of the same NMP operation, similar to the weight for an edge. To understand the affinity, we create N number of bins for each of the traits and place the pages in the intersection of both by considering the traits together. So the affinity space is $N(accesses) \times N(radix)$, which is further divided into four quadrants to produce a consolidated result in Figure 4c. Based on our study, a higher affinity indicates a harder problem, which poses greater challenges for finding a near-optimal solution. On the contrary, they also exhibit and offer a higher degree of scope for improvement. Please note that this is only one aspect of the complexities of the problem. Interestingly in our

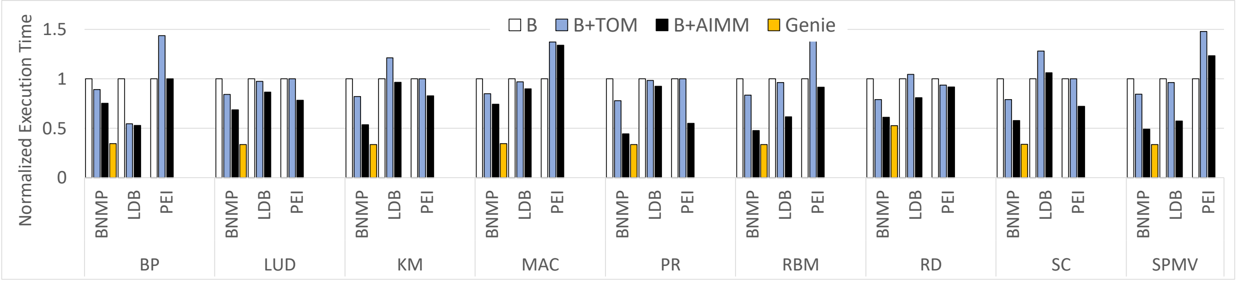


Fig. 5: Execution time for all the benchmarks, normalized individually with their basic techniques BNMP, LDB, and PEI respectively, which does not have any remapping support, and commonly referred as B in the graph. Execution time for the baseline implementations are compared to the system with remapping support, namely, TOM and AIMM, respectively. We have added an unrealistic setup (Genie) result to show a highly optimistic upper-bound.

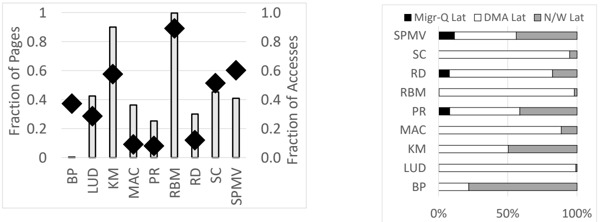
collection, we observe a balanced distribution of workloads in terms of their page affinity.

VI. Experimentation Results

A. Performance

1) Execution Time: Figure 5, shows the execution time for different applications under various system setup and support. It is evident that AIMM is effective at helping the NMP techniques to achieve better performance in almost all cases. We observe up to 55% improvement in execution time. With BNMP processing, the NMP operations are completely on the memory side, both TOM and AIMM boost the performance of BNMP. In general, TOM achieves around 15% to 20% performance improvement. AIMM secures improvement in execution time by around 50% on average across all the benchmarks. However for LDB, TOM did not improve much for most of the benchmarks. With LDB, AIMM improves execution time up to 43%, with no performance degradation observed except minor performance loss for *SC*. In the case of PEI, it is evident that for the majority of benchmarks TOM degrades the performance, whereas AIMM manages to achieve performance benefit around 10% to 20% (up to 42%) on average. For *SPMV* and *MAC*, both TOM and AIMM degrade the performance of PEI. There are several factors that play major roles in our system to drive system performance, such as operation per cycle, learning rate, utilization of migrated pages, path congestion and computation level parallelism. In addition, hop count in the memory network and row buffer hit rate in the memory cube can be considered as primary contributors for performance, depending on the nuances of individual cases. In the following, we discuss each of the techniques to justify their performance.

In the case of *PR* on BNMP, AIMM does not improve over TOM, which can mostly be attributed to the 3× computation distribution achieved by TOM over AIMM as shown on Figure 9a, in addition to very low migration page access for *PR* as shown in Figure 6. Figure 4a shows that *PR* has high number of pages that accessed small number of times, justifying low usage of migrated pages. Higher computation distribution improves NMP parallelism at the cost of extra communication if the computation node and operation destination are different. Low accesses to migrated pages diminishes the benefit of migration. On the other hand, AIMM achieves 50% better performance for *SPMV* that is ascribed to significant improvement in average hop count in Figure 9a, moderate fraction pages migrated (40%) and they constitute almost 60% of all the accesses as shown in Figure 6. The performance of *SC* with TOM and AIMM allows us to delve into discussion of the trade-off between computation distribution and hop count in the memory network, along with importance of distribution of the right candidate in the right place. Missing one of them may offset the benefit achieved by the other, as the case for *SC* with TOM. On the contrary, AIMM leads to a better performance than TOM with very moderate compute distribution and low hop count as shown in Figure 9a. Since LDB and PEI both tend to co-locate the sources belonging to an NMP operation, they leave much lesser chance than BNMP type techniques for further performance improvement using remapping like TOM or AIMM. The performance improvement is mainly achieved by optimizing the location of the destination pages with respect to the source operand pages.



(a)

(b)

Fig. 6: Migration Stats: (a) On the major axis we show the fraction of pages that are migrated for each of the applications using bars. On the minor axis it projects the fraction of total accesses that are happened on migrated pages, with diamond shaped markers. (b) Migration latency breakdown.

For estimating an upper bound of the execution time, we build a Genie (with unrealistic assumptions) that magically puts computation in the node with least congestion, and also instantly gets the operand pages in that node for immediate computations without any delay or overhead (not even memory access latency as shown in Figure 5, projects ≈12.5% standard deviation for improvement of Genie over AIMM, across applications.

B. Learning Convergence

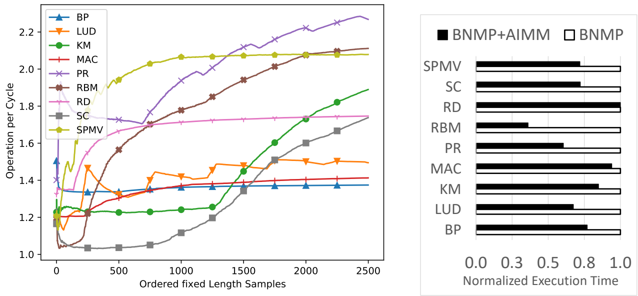
In Figure 7, we plot the timeline for OPC to show that AIMM progresses towards the goal for achieving high OPC as the time advances for each of the applications. Since the number of operations and so as the number of samples collected for each of the applications are different, we randomly choose a fixed number of points while preserving their original order of creation. The agent learns that the page migration has a long-term impact on the benefit of data co-location, whereas the computation migration balances load by keeping the computation and communication resource availability in consideration. *KM* and *SC* still show potential to improve OPC further given more episodes to run and other applications converged.

C. Migration

In Figure 6a, we depict the fraction of pages being migrated for each application on the major Y-axis and the fraction of total accesses requested that belong to a migrated page on the minor Y-axis. The fraction of pages migrated is an indicator of the migration coverage. For instance, in the case of *RBM* 100% pages are migrated, and almost all the migrated pages get accessed later. Pages in *RBM* are susceptible to experience high volume of migration as (1) small number of pages are accessed most of the execution time, (2) in a small fixed time window almost all the pages are being accessed. On the other hand, *BP* has huge memory residency and relatively small working set, which leads to a low fraction of page migration as compared to the total number of pages. Interestingly, the small number of migrated pages constitute almost 40% of the total accesses, which is a near ideal scenario as low number of page migrations has a small negative impact on performance, and a high number of accesses to the migrated pages can potentially improve the performance, provided the decision accuracy is high. The RL agent should learn from the wrong migration decisions through the feedback, as they incur performance overhead. Figure 6b shows migration latency breakdown, where, depending on the application, the migration latency is distributed in different proportions between DMA and HMC network. We observe average migration costs $\approx$ 800 cycles to 2000 cycles, depending on the network congestion. That is why we use non-blocking migration for all of our experiments. The migration/access across all the applications ranges from 0.001% to 0.054%, which can be categorized as low to moderate frequency.

D. Hop Count and Computation Utilization

In terms of system performance, hop count and computation distribution hold a reciprocal relation as reducing hop count improves the communication time, but results in computation under-utilization due to load imbalance across cubes. On the other hand, computation distribution may result in a high degree of hop count as concerned pages can only be in their respective memory cubes. Hence, a good balance between these factors is the key to achieving a near-optimal solution. Figure 9a shows that AIMM maintains a balance between the hop count and computation utilization. For instance, *PR* and *SC* individually achieves very high computation utilization which also leads to high average hop count. Assuming that the computation distribution decisions and corresponding locations are correct, the benefit gets diminished to 22% and 20% respectively possibly because of high hop count.



(a) (b)

Fig. 7: (a) Operation per cycle timeline. The X-axis is the sampled time and the Y-axis is the value for OPC. The graph is not monotonically increasing as OPC depends on several system parameters at runtime. (b) Normalized execution time for 8×8 mesh (shorter is better).

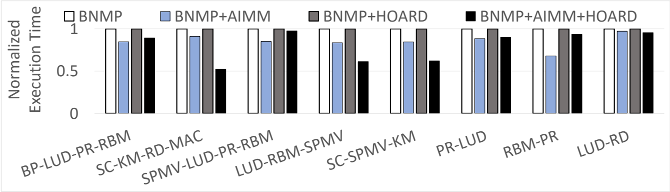
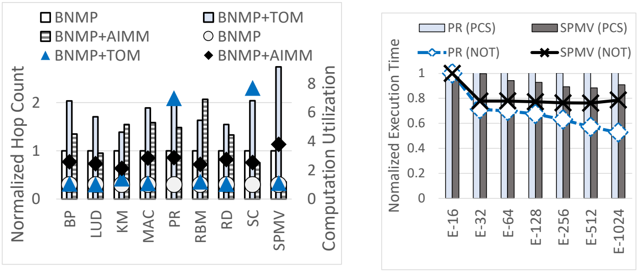


Fig. 8: Multi-process normalized execution time. BNMP and BNMP+HOARD are considered as two separate baselines and used for normalize BNMP+AIMM and BNMP+AIMM+HOARD results, respectively.

E. Scalability Study

In this subsection, we extend our experiments to study the impact of AIMM under a different underlying hardware configuration (8×8 mesh) as well as under highly diverse workloads together (2/3/4-processes). With a larger network, we expect to observe higher network impact on the memory access latency, and intend to show that AIMM can adapt to the changes in the system without any prior training on them. While choosing applications for multi-program workloads, we select diverse applications together based on our workload analysis, so that the RL agent experiences significant variations while trying to train and infer with them.

1) MCN Scaling: We observe that AIMM can sustain the changes in the underlying hardware by continuously evaluating them and without having any prior information. However, the amount of improvements for the applications are different than that in a 4×4 mesh. For instance, as shown in Figure 7b, with BNMP+AIMM, *RBM* observes more benefit over BNMP, than it observes in the 4×4 mesh. As we know larger networks are susceptible to higher network latency, however, they also offer higher capacity and potential throughput. In the case of *RBM*, AIMM could sustain throughput, whereas benefits for other benchmarks slightly offset, mostly because of higher network delay. Note that in terms of simulation with larger network size, we did not change the workload size. Tuning hyper parameters is left for future work.



(a)

(b)

Fig. 9: (a) Average Hop Count and Computation Utilization. Major Y-axis is shown in bars. (b) Sensitivity study: The bar graph shows the sensitivity of the benchmarks for different page-cache sizes (PCS), whereas the line graph show the sensitivity to the NMP-Op table (NOT) sizes.

2) Multi-program Workload: Figure 8 shows multi-process execution time. For a continuous learning environment like AIMM, multi-program workloads pose a tremendous challenge on the learning agent as well as on the system. For studying the impact of multi-program workloads, we consider two baselines. (1) BNMP, where the NMP operation tables, page info cache, etc., are shared and contended among all the applications. We leave the study of priority based allocation or partitioning for future study. (2) BNMP+HOARD, where at the page frame allocator level our modified version of HOARD helps to co-locate data for each process, preventing data interleaving across processes. We observe that for several application combinations (*SC-KM-RD-MAC*, *LUD-RBM-SPMV*, *SC-SPMV-KM*), HOARD and AIMM compliment each other to achieve 55% to 60% performance benefits.

F. Sensitivity Study

We study the system performance by varying the sizes of (1) page info cache, whose size is critical for passing system information to the agent and (2) NMP operation table, whose size is important to hold the entries for NMP operations, denial of which affects memory network flow. We choose two representative applications (*PR*, *SPMV*) to study their performance by varying one parameter while keeping the other one to its default value, mentioned in Table III. (3) We also tune the training hyper-parameters for optimum results.

1) Page-info cache size (PCS): Figure 9b shows that *PR* exhibits very minimal sensitivity to page cache size, whereas *SPMV* finds its sweet point while increasing the number of entry from 32 (E-32) to 64 (E-64). As a general trend, applications that get the most benefit from AIMM are more sensitive to the page cache size than others. Based on this study, we empirically decide the number of page cache entry as 256.

2) NMP table size (NMP-Op Tab): NMP table size sensitivity depends on several parameters such as the number of touched pages (Figure 4b), computation distribution, etc. In terms of average number of touched pages in an epoch, *SPMV* has around 10 pages on average in a time window. Figure 9a shows that *SPMV* has the highest computation distribution among the other applications. Combining these two pieces of data, it explains the reason for execution time saturation after 32 entries (E-32) for *SPMV*. On the other hand, *PR* has a very high demand for NMP operation table, as it has high average touched page count and low reuse rate. That is why *PR* shows monotonic improvement in execution time with the increase in the number of NMP table entries.

3) Training hyper-parameters: We study the sensitivity of the execution time with respect to replay-buffer size, frequency of training and learning rates. The replay-buffer size is varied 500× observing only up to 4% performance loss, which helped to optimize energy and area consumption without hurting performance much. As discussed earlier, training rate is controlled by the agent by its actions (action #6 and #7), we vary the upper and lower bound of it and observe a sweet point (execution time ±0.4% to ±13%). We apply the same strategy for learning rate as well, where we explore the spectrum to find the most suitable point (execution time ±0.1% to ±20%) and set as default for all the applications and experiments.

G. Area and Energy

In this section, we discuss the detailed area and energy aspect of our design. The implementation of the RL engine and migration management unit demands a separate provision in the chip. For estimating the area and energy, we model all the buffers and caches using Cacti [84], 45nm technology, since these components contribute to most of the area and energy overhead in the system, and use McPAT [85] for overall CMP area estimation. Overall area overhead as compared to 16 core Xeon processor is ≈0.2% and HMC area overhead is even more negligible.

1) Information Orchestration: We estimated the area for hardware registers and page information cache as part of the information orchestration system. Since the hardware registers occupy negligible area, we mostly focus on the page information cache of size 64KB, which occupies 0.23 mm^2 area. The estimated per access energy for page information cache is 0.05nJ, which is consumed every time the cache is updated and read.

2) Migration: For the migration system, we consider three data storing points as a major contributor for area, namely, NMP buffer (512B, 0.14mm^2), Migration queue (2KB, 0.04mm^2), and DMA buffers (1KB, 0.124mm^2). It is worth noting that, depending on the organization and access method, the buffer and cache peripherals change significantly and so does their area. The per access energy consumption by these components are 0.122nJ, 0.02689nJ, 0.1062nJ, respectively. Page wise energy in the average case is composed of DMA leakage energy (15.51nJ) for waiting 500 cycles and the network cost for the whole page-frame transfer (574.44nJ), in total ≈ 600nJ.

3) RL Agent: For estimating the area and energy consumed by the RL agent, we focus on their major source of energy consumption that can easily be modeled as cache like structures, namely weight matrix (603KB, 2.095mm^2), replay buffer (≈512KB, 3.17mm^2), and state buffer (576B, 0.12mm^2). Their per access energy is estimated as 0.244nJ, 0.316nJ, and 0.106nJ, respectively.

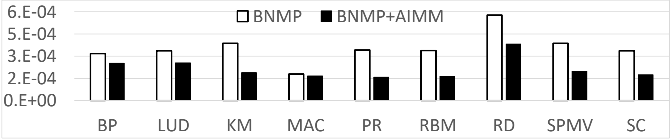


Fig. 10: Energy-delay product (lower is better).

4) Network and Memory: Since the migrations are realized by actually sending pages through the memory network with the help of DMA, we also estimate the network and memory cube energy consumption by assuming 5pJ/bit/hop [86] and 12pJ/bit/access [5] for the network and memory, respectively.

5) Overall Dynamic Energy: In our overall dynamic energy study, we include energy consumed by (1) only additional AIMM hardware, (2) memory network energy, and (3) memory cube energy, as major contributors to energy consumption in our framework. In Figure 10 we project Energy-Delay Product (EDP) as an overhead estimation metric, comparing the baseline and AIMM setup across the benchmarks. The EDP is roughly equivalent to the reciprocal of MIPS^2/Watt [87], where MIPS stands for Million Instructions per second and Watt (Joul/second) is the unit of power. In our case we replace instructions with NMP operations. The delay is the runtime, which is converted to seconds, assuming 2.66 GHz processor clock frequency. Overall EDP shows that, even though AIMM has some energy overhead over the baseline, it is still beneficial, as (except *MAC*) we observe that all the benchmarks' EDP is significantly lower than baseline.

VII. Case Study: Scalable HBM-PIM

In this section, we discuss a case study and potential solution for addressing the capacity scalability issue with HBM-PIM [24]. It is pertinent to have a scalable solution, as many modern applications (good candidates for NMP) like, embedding lookup for embedding reduction [88], DNN kernel handling large input [89], [90], and other machine learning kernels used in our paper demands huge memory for handling large inputs. To evaluate the scope of AIMM in hierarchical HBM-PIM network, we develop a Transaction Level Model (TLM) [91], capturing high level functional behaviors of the hierarchical HBM-PIM.

	Benchmarks					
	BP	LUD	KM	MAC	SPMV	DNN-KER [90]
Norm Exec-Time	0.99	0.65	0.36	0.94	0.29	0.40

TABLE V: Normalized execution time.

To overcome the memory capacity scaling limitation along with computation bandwidth, we propose a hierarchical network of HBM-PIMs enabled with smart switches and connected through smart network interfaces (SMART-NIC). Considering current HBM-PIM as one tile, we connect 4 tiles (scaled up to 128GB) and a host CMP using a smart switch (5×5 ports, NMP computation enabled) with 100GB/s PCIe bus. The processor chip in the tile is used only for NMP computation and memory commands are scheduled by MCs, whereas the host CMP is used for operation offloading. The TLM model connects the modules using generic port, and uses generic payload and AXI4-Stream protocol [92] for communication; different constant latencies/buffer sizes are used to model memory access latency, computation delay, link accesses latency, reflecting their corresponding operating frequencies, bandwidth and pipeline behavior. We integrate this model into AIMM-sim. We provide occupancy for PEs (in switches/HBMs), read/write queues in HBMs, and generic ports as information to construct the states. Actions still dictate migrations (estimated overhead for page migration is less than 10% of execution time, is not considered) and the reward is constructed using OPC. The experiments are done with 10 million NMP operations. The results for a subset of applications are shown in Table V. In our preliminary study we observe large variations in the improvement over baseline, ranging from none to 71% in execution time, possibly ascribed to optimization of computation location (external switch/tile switch/HBM-PIM) through improved data mapping. Even though we leave detailed results analysis for future work, we can see clear scope for AIMM in the hierarchical memory network with heterogeneous link latency and bandwidth.

VIII. Conclusions and Future Directions

Careful articulation of data placement in the physical memory cube network (MCN) becomes imperative with the advent of Near-Memory Processing (NMP) in the big data era. In addition, scheduling computation for both resource utilization and data co-location in large-scale NMP systems is even more challenging than ever before. We propose AIMM, which is shown to be effective in assisting the existing NMP techniques mapped on MCN, by remapping their computation and data for improving resource utilization and optimizing communication overhead. Driven by the application's dynamic memory access behavior and the intractable size of data mapping decision space, AIMM uses Reinforcement Learning techniques as an approximate solution for optimization of the data and computation mapping problem. We project our technique as a plug-and-play module to be integrated with diverse NMP systems. The comprehensive experimentation shows significant performance improvement with up to 55% for single-program workloads and up to 50% for multi-program workloads over baseline NMP.

Expanding AIMM for facilitating data and command mapping for *Processing Using Memory* may alleviate the burden of handpicking different mappings for different underlying hardware, and further optimize its performance by continuously improving the mapping in orderly fashion. We believe AIMM-sim will inspire and enable integrated machine learning and hardware simulation research on AI in hardware.

References

[1] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "Cortexsuite: A synthetic brain benchmark suite.," in *IISWC*, pp. 76–79, 2014.

[2] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores," in *International Symposium on Workload Characterization (IISWC)*, pp. 44–55, IEEE Computer Society, 2015.

[3] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[4] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[5] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Hot Chips 23 Symposium (HCS)*, pp. 1–24, IEEE, 2011.

[6] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, *et al.*, "25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pp. 432–433, IEEE, 2014.

[7] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *International Symposium on Computer Architecture (ISCA)*, pp. 336–348, IEEE, 2015.

[8] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *International Symposium on Computer Architecture (ISCA)*, pp. 105–117, IEEE, 2015.

[9] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 457–468, 2017.

[10] J. Huang, R. R. Puli, P. Majumder, S. Kim, R. Boyapati, K. H. Yum, and E. J. Kim, "Active-Routing: Compute on the Way for Near-Data Processing," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 674–686, IEEE, 2019.

[11] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *International Symposium on Computer Architecture (ISCA)*, pp. 204–216, IEEE Press, 2016.

[12] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[13] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-Centric System Interconnect Design with Hybrid Memory Cubes," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 145–156, IEEE Press, 2013.

[14] J. Zhan, I. Akgun, J. Zhao, A. Davis, P. Faraboschi, Y. Wang, and Y. Xie, "A Unified Memory Network Architecture for In-Memory Computing in Commodity Servers," in *International Sympoium on Microarchitecture (MICRO)*, pp. 1–14, IEEE, 2016.

[15] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pp. 32–41, 2000.

[16] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3d-stacked dram," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 131–143, 2015.

[17] Y. Liu, X. Zhao, M. Jahre, Z. Wang, X. Wang, Y. Luo, and L. Eeckhout, "Get out of the valley: power-efficient address mapping for gpus," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 166–179, IEEE, 2018.

[18] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira, "Compiler support for selective page migration in numa architectures," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), p. 369–380, Association for Computing Machinery, 2014.

[19] B. Goglin and N. Furmento, "Enabling high-performance memory migration for multithreaded applications on linux," in *2009 IEEE International Symposium on Parallel Distributed Processing*, pp. 1–9, 2009.

[20] M. Chiang, S. Tu, W. Su, and C. Lin, "Enhancing inter-node process migration for load balancing on linux-based numa multicore systems," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 02, pp. 394–399, 2018.

[21] Z. Duan, H. Liu, X. Liao, H. Jin, W. Jiang, and Y. Zhang, "Hinuma: Numa-aware data placement and migration in hybrid memory systems," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pp. 367–375, 2019.

[22] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 185–195, IEEE, 2013.

[23] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70, IEEE, 2020.

[24] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, *et al.*, "25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2 tflops programmable computing unit using bank-level parallelism, for machine learning applications," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 350–352, IEEE, 2021.

[25] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *International Symposium on Microarchitecture (MICRO)*, pp. 273–287, ACM, 2017.

[26] V. Seshadri and O. Mutlu, "Simple operations in memory to reduce data movement," in *Advances in Computers*, vol. 106, pp. 107–166, Elsevier, 2017.

[27] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast bulk bitwise and and or in dram," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 127–131, 2015.

[28] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 288–301, IEEE, 2017.

[29] A. Yazdanbakhsh, C. Song, J. Sacks, P. Lotfi-Kamran, H. Esmaeilzadeh, and N. S. Kim, "In-dram near-data approximate acceleration for gpus," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pp. 1–14, 2018.

[30] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie, "ipim: Programmable in-memory image processing accelerator using near-bank architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 804–817, IEEE, 2020.

[31] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 383–396, IEEE, 2018.

[32] J. Huang, P. Majumder, S. Kim, T. Fulton, R. R. Puli, K. H. Yum, and E. J. Kim, "Computing en-route for near-data processing," *IEEE Transactions on Computers*, vol. 70, no. 6, pp. 906–921, 2021.

[33] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *SIGPLAN Not.*, vol. 35, p. 117–128, Nov. 2000.

[34] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management* (H. G. Baler, ed.), (Berlin, Heidelberg), pp. 1–116, Springer Berlin Heidelberg, 1995.

[35] K. C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, p. 623–624, Oct. 1965.

[36] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Oreilly amp; Associates Inc, 2005.

[37] J. Corbet, "LKML: Transparent huge pages in 2.6.38," Jan. 2011.

[38] A. Panwar, S. Bansal, and K. Gopinath, "Hawkeye: Efficient fine-grained os support for huge pages," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 347–360, Association for Computing Machinery, 2019.

[39] R. Courts, "Improving locality of reference in a garbage-collecting memory management system," *Commun. ACM*, vol. 31, p. 1128–1138, Sept. 1988.

[40] L. P. Deutsch and D. G. Bobrow, "An efficient, incremental, automatic garbage collector," *Commun. ACM*, vol. 19, p. 522–526, Sept. 1976.

[41] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Commun. ACM*, vol. 26, p. 419–429, June 1983.

[42] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox, "Numa policies and their relation to memory architecture," *ACM SIGOPS Operating Systems Review*, vol. 25, no. Special Issue, pp. 212–221, 1991.

[43] "Numa locality,"

[44] A. C. Yao, "An analysis of a memory allocation scheme for implementing stacks," *SIAM Journal on Computing*, vol. 10, no. 2, pp. 398–403, 1981.

[45] C. P. Ribeiro, J. Mehaut, A. Carissimi, M. Castro, and L. G. Fernandes, "Memory affinity for hierarchical shared memory multiprocessors," in *2009 21st International Symposium on Computer Architecture and High Performance Computing*, pp. 59–66, 2009.

[46] C. Lameter, "Numa (non-uniform memory access): An overview," *Queue*, vol. 11, 07 2013.

[47] T. Baruah, Y. Sun, A. T. Dinçer, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Griffin: Hardware-software support for efficient page migration in multi-gpu systems," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 596–609, IEEE, 2020.

[48] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding tlb shootdowns through self-invalidating tlb entries," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 273–287, IEEE, 2017.

[49] M. Oskin and G. H. Loh, "A software-managed approach to die-stacked dram," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 188–200, IEEE, 2015.

[50] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "Unified instruction/translation/data (unitd) coherence: One protocol to rule them all," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, IEEE, 2010.

[51] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 340–349, IEEE, 2011.

[52] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[53] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[54] J.-Y. Won, X. Chen, P. Gratz, J. Hu, and V. Soteriou, "Up by their bootstraps: Online learning in artificial neural networks for cmp uncore power management," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 308–319, IEEE, 2014.

[55] T.-R. Lin, D. Penney, M. Pedram, and L. Chen, "A deep reinforcement learning framework for architectural exploration: A routerless noc case study," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 99–110, IEEE, 2020.

[56] J. Yin, S. Sethumurugan, Y. Eckert, C. Patel, A. Smith, E. Morton, M. Oskin, N. E. Jerger, and G. H. Loh, "Experiences with ml-driven design: A noc case study," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 637–648, IEEE, 2020.

[57] K. Wang, A. Louri, A. Karanth, and R. Bunescu, "Intellinoc: A holistic design framework for energy-efficient and reliable on-chip communication for manycores," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, IEEE, 2019.

[58] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 285–297, IEEE, 2015.

[59] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 39–50, 2008.

[60] B. H. Ahn, P. Pilligundla, and H. Esmaeilzadeh, "Reinforcement learning and adaptive sampling for optimized dnn compilation," *arXiv preprint arXiv:1905.12799*, 2019.

[61] S.-C. Kao, G. Jeong, and T. Krishna, "Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 622–636, IEEE, 2020.

[62] N. Wu, L. Deng, G. Li, and Y. Xie, "Core placement optimization for multi-chip many-core neural network systems with reinforcement learning," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 2, pp. 1–27, 2020.

[63] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 855–864, 2016.

[64] A. Demir, E. Çilden, and F. Polat, "Landmark based reward shaping in reinforcement learning with hidden states," in *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pp. 1922–1924, 2019.

[65] J. Ferret, R. Marinier, M. Geist, and O. Pietquin, "Self-attentional credit assignment for transfer in reinforcement learning," *arXiv preprint arXiv:1907.08027*, 2019.

[66] X. Guo, *Deep learning and reward design for reinforcement learning*. PhD thesis, 2017.

[67] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "Fa3c: Fpga-accelerated deep reinforcement learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 499–513, Association for Computing Machinery, 2019.

[68] J. Su, J. Liu, D. B. Thomas, and P. Y. Cheung, "Neural network based reinforcement learning acceleration on fpga platforms," *SIGARCH Comput. Archit. News*, vol. 44, p. 68–73, Jan. 2017.

[69] Z. Yan, J. Veselỳ, G. Cox, and A. Bhattacharjee, "Hardware translation coherence for virtualized systems," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 430–443, 2017.

[70] M. Plappert, "keras-rl." https://github.com/keras-rl/keras-rl, 2016.

[71] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[72] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE micro*, vol. 40, no. 3, pp. 20–29, 2020.

[73] D.-I. Jeon and K.-S. Chung, "Cashmc: A cycle-accurate simulator for hybrid memory cube," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 10–13, 2016.

[74] N. Jiang, G. Michelogiannakis, D. Becker, B. Towles, and W. J. Dally, "Booksim 2.0 user's guide," *Standford University*, p. q1, 2010.

[75] A. Singh, D. Carnelli, A. Falay, K. Hofstra, F. Licciardello, K. Salimi, H. Santos, A. Shokrollahi, R. Ulrich, C. Walter, *et al.*, "26.3 a pin-and power-efficient low-latency 8-to-12gb/s/wire 8b8w-coded serdes link for high-loss channels in 40nm technology," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 442–443, IEEE, 2014.

[76] D. I. Jeon and K. S. Chung, "CasHMC: A Cycle-Accurate Simulator for Hybrid Memory Cube," *IEEE Computer Architecture Letters*, vol. 16, pp. 10–13, Jan 2017.

[77] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *IEEE International Symposium on Workload Characterization (IISWC'10)*, pp. 1–11, IEEE, 2010.

[78] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization (IISWC)*, pp. 44–54, IEEE Computer Society, 2009.

[79] "Multiple features data set," *https://archive.ics.uci.edu/ml/datasets/Multiple+Features*.

[80] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[81] G. E. Hinton, "Boltzmann machine," *Scholarpedia*, vol. 2, no. 5, p. 1668, 2007.

[82] "Netflix public data sets," *https://github.com/Netflix/vmaf/blob/master/resource/doc/datasets.md*.

[83] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[84] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[85] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 469–480, 2009.

[86] M. Poremba, I. Akgun, J. Yin, O. Kayiran, Y. Xie, and G. H. Loh, "There and Back Again: Optimizing the Interconnect in Networks of Memory Cubes," in *International Symposium on Computer Architecture (ISCA)*, pp. 678–690, ACM, 2017.

[87] S. Kaxiras and M. Martonosi, "Computer architecture techniques for power-efficiency," *Synthesis Lectures on Computer Architecture*, vol. 3, no. 1, pp. 1–207, 2008.

[88] Y. Lee, S. H. Seo, H. Choi, H. U. Sul, S. Kim, J. W. Lee, and T. J. Ham, "Merci: efficient embedding reduction on commodity hardware via sub-query memoization," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 302–313, 2021.

[89] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

[90] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 461–475, ACM, 2018.

[91] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (IEEE Cat. No. 03TH8721)*, pp. 19–24, IEEE, 2003.

[92] "AXI reference guide," *https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf*, 2011.