2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Processing-in-Memory for Energy-efficient Neural Network Training: A Heterogeneous Approach

Jiawen Liu*[‡], Hengyu Zhao*[†], Matheus Almeida Ogleari[♯], Dong Li[‡], Jishen Zhao[†]
[†]*University of California, San Diego* [‡]*University of California, Merced* [♯]*University of California, Santa Cruz*
[†]*{h6zhao, jzhao}@ucsd.edu* [‡]*{jliu265, dli35}@ucmerced.edu* [♯]*mogleari@ucsc.edu*

***Abstract*—Neural networks (NNs) have been adopted in a wide range of application domains, such as image classification, speech recognition, object detection, and computer vision. However, training NNs – especially deep neural networks (DNNs) – can be energy and time consuming, because of frequent data movement between processor and memory. Furthermore, training involves massive fine-grained operations with various computation and memory access characteristics. Exploiting high parallelism with such diverse operations is challenging. To address these challenges, we propose a software/hardware co-design of heterogeneous processing-in-memory (PIM) system. Our hardware design incorporates hundreds of fix-function arithmetic units and ARM-based programmable cores on the logic layer of a 3D die-stacked memory to form a heterogeneous PIM architecture attached to CPU. Our software design offers a programming model and a runtime system that program, offload, and schedule various NN training operations across compute resources provided by CPU and heterogeneous PIM. By extending the OpenCL programming model and employing a hardware heterogeneity-aware runtime system, we enable high program portability and easy program maintenance across various heterogeneous hardware, optimize system energy efficiency, and improve hardware utilization.**

I. Introduction

Neural networks (NNs) have been adopted by a wide range of application domains, such as computer vision, speech recognition, and natural language processing. Today, NN models employ increasingly larger number of parameters and data sets. For example, VGG [1] and AlexNet [2] employ 138M and 61M parameters for image classification, respectively. Training such complex models demands immense computation and memory resources, energy and time. One critical energy and performance bottleneck when training NN is data movement in systems. As NN models are becoming deeper and larger, the data volume and the pressure on the runtime system to support data intensive operations substantially increase. Existing research efforts use low-precision data [3] or prune NN models [4]. Yet, these efforts impose the difficulty of quantifying the impact of model simplification on NN model accuracy; They do not fundamentally address the data movement problem in NN model training.

Recent development of processing-in-memory (PIM) techniques have been explored as a promising solution to tackle the data movement challenge in various applications [5, 6]. We profile various NN training workloads and reveal that such workloads have diverse memory access patterns, computation intensity, and parallelism (Section II). As a result, NN training can significantly benefit from heterogeneous PIM – which incorporates both fixed-function logic and programmable cores in the memory – to achieve optimal energy efficiency and balance between parallelism and programmability. However, such heterogeneous PIM architecture introduces multiple challenges in the programming method and runtime system.

*These two authors contributed equally to this paper as the first authors.

First, programming PIMs to accelerate NN training is non-trivial. Today, the common machine learning frameworks, such as TensorFlow [7], Caffe2 [8], heavily rely on a variety of implementations for NN operations on various hardware, and use a middleware to integrate those operations to provide hardware transparency to the user. Such a software design can place high burden on system programmers, because of the increasing hardware heterogeneity and difficulty for program maintenance. Most previous PIM software interfaces [5, 6, 9] require programmers to have the detailed knowledge of underlying hardware. In order to improve productivity and ease-of-adoption of PIM-based NN training accelerators, we need to develop a programming method that maximizes code reuse without asking the programmer to repeatedly program on different PIMs.

Second, combining fixed-function logics and programmable cores in PIM further complicates the software design. Fixed-function and programmable PIMs employ vastly different programming models: Fixed-function PIMs employ ISA-level instructions accessed via assembly-level intrinsics or via library calls; Programmable PIMs employ standard programming paradigms, such as threading packages or GPGPU programming interfaces [10]. As discussed in recent studies [10], most previous PIM designs adopt homogeneous PIM architectures – with either fixed-function or programmable PIMs – which allows a simplified software interface design. But with heterogeneous PIM, it is critical to design a unified programming model that can accommodate both PIM components.

Finally, the scale of operations in NN training can lead to unbalanced hardware utilization. Ideally, we want to achieve high utilization of PIMs without violating the dependency requirement among NN training operations, by exploiting abundant operation-level parallelism across the host processor and PIMs. However, it can be difficult to achieve so in such a heterogeneous system by pure hardware scheduling, because of the complexity of tracking operation dependency and synchronization. Furthermore, NN training typically adopts a large amount (e.g., tens of thousands) of iterative steps and hundreds of operations per step. Operation dependency across the massive amount of steps and operations can impose synchronization overhead and decrease hardware utilization, when operations are running on multiple computing devices.

Our goal in this paper is to design a PIM-based NN training acceleration system that can efficiently accelerate unmodified training models written on widely-used machine learning frameworks (e.g., TensorFlow). To achieve our goal, we propose a software/hardware co-design of a heterogeneous PIM framework. Our design consists of three components. First, we adopt a heterogeneous PIM architecture, which integrates both fixed-function logics and programmable cores in 3D die-stacked main memory. Second, we extend the OpenCL [11] programming model to address the programming challenges. The programming model maps the host CPU and heterogeneous PIMs onto OpenCL's platform model and extends OpenCL's execution and memory models for efficient runtime scheduling. Finally, we propose a runtime system, which maximizes PIM hardware utilization and NN-operation-level parallelism. This paper makes the following contributions:

- We developed a profiling framework to characterize NN training models written on TensorFlow. We identify the heterogeneity requirements across the operations of various NN training workloads. Based on our profiling results, we identify opportunities and key challenges in the software design for efficiently accelerating NN training using PIMs.
- We develop a heterogeneous PIM architecture and demonstrate the effectiveness of such an architecture for training NN models.
- We propose an extension to OpenCL programming model in order to accommodate the PIM heterogeneity and improve the program maintainability of machine learning frameworks.
- We propose a runtime system to dynamically map and schedule NN operations on heterogeneous PIM, based on dynamic profiling of NN operations.

II. Background and Motivation

We motivate our software/hardware coordinated design by discussing the challenges of accelerating machine learning training workloads. We employ three widely used CNN training models – VGG-19 [1], AlexNet [2], and DCGAN [12] – as examples in this section. However, our observations can also be applied to various other training workloads (Sections VI).

A. NN Training Characterization

In order to understand the characteristics of NN training workloads, we develop a profiling framework (Figure 1) by leveraging TensorBoard [13] and Intel VTune [14] to collect software and hardware counter information of training operations. Measuring the number of main memory accesses of individual operations during training can be inaccurate due to the extra cache misses imposed by simultaneously executing operations. As such, we disable inter-operation parallelism to ensure characterization accuracy of individual operations.

Table I illustrates our profiling results of top five most time-consuming and memory-intensive operations, respectively, with three training models. Each model has tens of different types

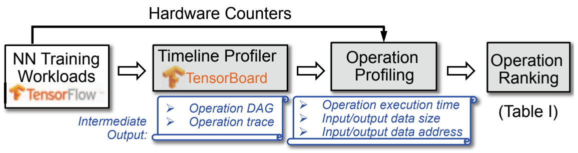


Fig. 1: Our profiling framework for profiling NN training workloads in TensorFlow.

of operations and requires thousands of iterative steps to train; In each step, each type of operation can be invoked up to tens of times. We only show results within one training step. But the characteristics remain stable across training steps.

We make three key observations. First, only several operations dominate training execution time. For example, top five operations in VGG-19 model consume over 95% of total execution time. Second, the most time-consuming operations are also the most memory intensive. In fact, the top five most time-consuming operations contribute to over 98% of total main memory accesses across all three models. We further classify operations into four classes, shown in Figure 2. The first class of operations is compute intensive, and does not have to be offloaded to PIMs, but we can offload them when there are idling hardware units in PIMs. The second class of operations is our target to offload to PIMs. The third class is unusual, and the fourth class does not have big performance impact on model training. *The above two observations motivate us to adopt a PIM architecture to accelerate NN training in order to reduce data movement between the host processor and the main memory.*

Execution Time	Memory Access %	Example Operations
Long	Low	*Conv2D* in VGG-19
Long	High	*Conv2DBackpropFilter* in VGG-19
Short	High	*Slice* in DCGAN
Short	Low	*Reshape* in AlexNet

Fig. 2: Four categories of NN training operations.

Third, time-consuming and memory-intensive operations require heterogeneous computation types. It appears that many of such operations are multiplication and addition (e.g., MatMul) or can be decomposed so (e.g., Conv2D). This is inline with previous works on machine learning acceleration [5, 6]. Yet, significant amount of top time-consuming and memory-intensive operations cannot simply be implemented by pure multiplication and addition. For instance, Relu is an activation function that incorporates conditional statement; MaxPool is a sample-based discretization process; ApplyAdam is a first-order gradient-based optimization of stochastic objective functions. Complex operations, such as Conv2DBackpropFilter and Conv2DBackpropInputs, include other logic and computations beyond multiplication and addition. Such non-multiply-add operations can consume over 40% of total execution time. Furthermore, studies on modern multi-tenancy [15] and multimodel training [16] workloads also demonstrate such heterogeneous computation requirement. *This observation motivates us to adopt a heterogeneous PIM architecture that combines fixed-function logic and programmable cores.*

Most previous works on PIM adopt either fixed-function [5]

TABLE I: Operation profiling results for three neural network models. "CI"= computation intensive; "MI"=memory intensive.

VGG-19					
Top 5 CI Ops	Execution Time(%)	#Invocation	Top 5 MI Ops	#Main Memory Access(%)	#Invocation
1. Conv2DBackpropFilter	40.15	16	1. Conv2DBackpropFilter	42.52	16
2. Conv2DBackpropInput	32.68	15	2. BiasAddGrad	35.68	16
3. BiasAddGrad	11.92	16	3. Conv2DBackpropInput	21.06	15
4. Conv2D	10.34	16	4. MaxPoolGrad	0.22	16
5. MaxPoolGrad	1.49	16	5. Relu	0.14	19
Other 13 ops	3.37	232	Other 13 ops	0.38	229
AlexNet					
Top 5 CI Ops	Execution Time(%)	#Invocation	Top 5 MI Ops	#Main Memory Access(%)	#Invocation
1. Conv2DBackpropFilter	33.64	5	1. BiasAddGrad	44.64	3
2. Conv2DBackpropInput	33.46	4	2. Conv2DBackpropInput	36.61	4
3. MatMul	13.54	6	3. Conv2DBackpropFilter	14.79	5
4. Conv2D	10.48	5	4. Relu	1.20	8
5. BiasAddGrad	4.62	3	5. Conv2D	0.46	5
Other 13 ops	4.26	121	Other 13 ops	2.30	119
DCGAN					
Top 5 CI Ops	Execution Time(%)	#Invocation	Top 5 MI Ops	#Main Memory Access(%)	#Invocation
1. Conv2DBackpropFilter	19.98	4	1. Conv2DBackpropFilter	37.21	4
2. Conv2DBackpropInput	17.18	4	2. Conv2DBackpropInput	28.09	4
3. MatMul	14.28	12	3. Slice	17.18	14
4. Conv2D	10.53	4	4. Conv2D	5.45	4
5. Mul	9.89	84	5. Mul	2.22	84
Other 47 ops	28.14	821	Other 47 ops	9.85	819

or programmable [6] computation components in the logic layer of 3D die-stacked memory. In the following, we discuss feasibility, challenges, opportunities of accelerating NN training with software/hardware co-design of heterogeneous PIM.

B. *Feasibility of Heterogeneous PIM Architecture*

The logic layer of 3D memory stacks has area, power, and thermal limitations. But previous studies demonstrated the feasibility of adopting both fixed-function and programmable PIMs, while meeting these constraints [17]. We adopt similar methodologies to ensure the feasibility of our architecture implementation (Section IV).

C. *Software Design Challenges and Opportunities*

There are three challenges for the software design (introduced in Section I): (1) How do we enable high productivity of system programmers and ease-of-adoption of PIM-based NN training accelerators? (2) How do we develop a unified programming model that can efficiently accommodate the host processor, fixed-function PIMs, and programmable PIMs? (3) How do we balance hardware utilization at runtime?

One candidate baseline programming model is OpenCL [11], which is widely used in accelerator-based heterogeneous computing platforms (e.g., GPU and FPGA). We adopt OpenCL, due to its portability, expressiveness, and ability to enable high programming productivity to support programming on heterogeneous systems (details are discussed in Section III-B). However, it is not straightforward to adopt OpenCL for NN model training on the heterogeneous PIM architecture. (1) How do we map the platform model of OpenCL to the heterogeneous PIM architecture? (2) Given the execution model of OpenCL with limited considerations on hardware utilization, how do we make the best use of CPU (the host processor) and different types of PIMs? (3) Given the memory model of OpenCL with limited considerations on synchronization between hardware units, how do we meet the requirement of frequent synchronizations from NN operations?

Trade-offs between parallelism and programmability. Fixed-function PIMs typically offer high computation parallelism by executing fine-grained, simple operations distributed across massive amount of logic units. However, they are less flexible than programmable PIMs that can be programmed to accommodate a large variety of operations. Furthermore, fixed-function PIMs can impose high performance overhead by (i) frequent operation-spawning and (ii) host-PIM synchronization. Programmable PIMs typically execute coarse-grained code blocks with less frequent host-PIM synchronization. However, the limited number of computational units in programmable PIMs can lead to much lower parallelism than in fixed-function PIMs.

Opportunities in runtime system scheduling. Substantial opportunities exist in leveraging system-level software to optimize resource sharing among various system components. The heterogeneity of our architecture introduces requirements on scheduling model-training operations across the host processor (CPU), fixed-function PIMs and programmable PIMs, based on the dynamic utilization of compute resources on these system components. Yet, we observe that NN training workloads tend to have repeatable (hence predictable) computation behavior over the execution time. As such, system software can accurately predict and dynamically schedule the operations by profiling the resource utilization of various compute elements in the first few steps of modeling training. Such dynamic profiling-based scheduling can achieve the best utilization of computation resources, while improving energy efficiency.

D. *CPU vs. GPU – Where to Attach Heterogeneous PIMs?*

Today, NN-training workloads can be executed on both CPU- and GPU-based systems. Recent silicon interposer technology allows both types of systems to adopt 3D die-stacked memories closely integrated with logic components. For example, modern GPU device memories [18] are implemented by high-bandwidth memory technology. High-end CPU servers integrate high-bandwidth memories using the DRAM technology adopted from hybrid memory cubes.

Our heterogeneous PIMs are logic components closely integrated with die-stacked memories. Therefore, they are generally applicable to both CPU or GPU systems. However, this paper focuses on the software design for heterogeneous PIMs attached on CPU systems, due to the constraint of current GPU systems. Today, GPU systems often fuse and organize computation kernels into NN layers rather than fine-grained operations, because of the inefficiency of compute preemption and thread scheduling. This significantly limits the flexibility of operation scheduling on GPU.

The NVIDIA Volta GPU provides certain support for fine-grained acceleration of NN training operations, yet only available with limited number of threads. Modern CPU systems are easy to access and program; this enables easy-to-adopt and flexible programming abstraction and system library functions.

III. Design

To address the aforementioned challenges, we propose a software/hardware co-design of heterogeneous PIM framework to accelerate NN training. Our design consists of a heterogeneous PIM architecture, an extended OpenCL programming model, and a runtime system. Figure 3 depicts our architecture configuration. Figure 4 shows the process of building and executing NN training with our software framework. Given an OpenCL kernel to implement an operation, our system extracts code sections from the kernel and compiles them into a set of binaries to run on CPU, programmable PIM, and fixed-function PIMs, respectively. After the training workload starts to execute, our runtime scheduler profiles the first step of training to obtain operation characterization. It then performs dynamic scheduling of operations across CPU, programmable PIM, and fixed-function PIMs in the rest of training steps. Our runtime system incorporates two key components: (i) an operation-pipeline scheme, which allows multiple NN operations to co-run on PIMs to improve hardware utilization and (ii) a recursive operation-execution scheme, which allows the programmable PIM to call fixed-function PIMs to improve hardware utilization and avoid frequent synchronization between CPU and PIMs.

Software/hardware co-design principles. Our software design supports our hardware configuration in the following manner. First, our software design offers a portable programming model across the host processor, fixed-function PIMs, and the programmable PIM. Our programming model provides a unified abstract to program various PIMs, which need to be programmed in separate manners in conventional systems. Our runtime scheduling scheme effectively optimizes PIM hardware utilization. Our runtime system also enables recursive calls between the programmable PIM and fixed-function PIMs. Our architecture design supports our software design in two ways: our heterogeneous PIM architecture enables efficient NN training acceleration by exploiting the heterogeneous characteristics of software operations; We employ a set of hardware registers to track PIM hardware utilization information, which is required by our runtime scheduling.

A. Heterogeneous PIM Architecture

To accommodate various types of operations that are likely to execute on PIMs, we adopt a heterogeneous PIM architecture consisting of (i) a programmable PIM, which is an ARM core and (ii) massive fixed-function PIMs, which are adders and multipliers distributed across all memory banks. While our design can be used with various 3D die-stacked memory devices, we employ a 32-bank memory stack (where a bank is a vertical slice in the stack) as an example in this paper. Figure 3 depicts our architecture configuration. Section IV describes hardware implementation details.

B. Programming Model for Heterogeneous PIM

We extend the OpenCL programming model to program the heterogeneous PIM. OpenCL has been widely employed to enable program portability across accelerator-based, heterogeneous computing platforms (e.g., GPU and FPGA). We use OpenCL because of the following reasons. First, by treating the fixed-function PIMs and programmable PIM as accelerators, the semantics of OpenCL naturally fit into the heterogeneous PIM environment. Second, writing a program for the heterogeneous PIM based on an abstract and unified hardware model in OpenCL, the programmer can write the program just once but run it on a variety of PIMs. Therefore, by using OpenCL, we can hide hardware variety of the heterogeneous PIM from system programmers, improve their productivity, and enable code portability.

Other programming models, such as OpenACC [19, 20] and OpenMP [21], can also hide hardware heterogeneity and reduce programmers' burden. However, these are higher-level programming models, which rely on compilers to transform programs into a lower-level programming model, such as OpenCL, to enable code portability. We focus on OpenCL in our study, because it provides a foundation for those higher-level programming models.

Overview of our programming model. Table II summarizes our extension to OpenCL. Our platform model includes multiple types of heterogeneous devices. Such platform model is driven by the characteristics of NN training operations. Our execution model adds (i) recursive kernel invocation to enable kernel invocation between PIMs to support complex NN operations (e.g., Conv2DBackpropFilter) and (ii) operation pipeline to improve hardware utilization for small NN operations with limited parallelism (e.g., Slice). Finally, we extend the memory model to support a single global memory shared between the host processor and accelerators. We also add explicit synchronization across different PIMs and CPU (host processor) to enforce execution orders across NN operations.

OpenCL background. The existing OpenCL adopts a host-accelerator platform model as shown in Figure 5(a). A host processor connects to one or more compute devices (i.e., accelerators). A compute device is divided into one or more

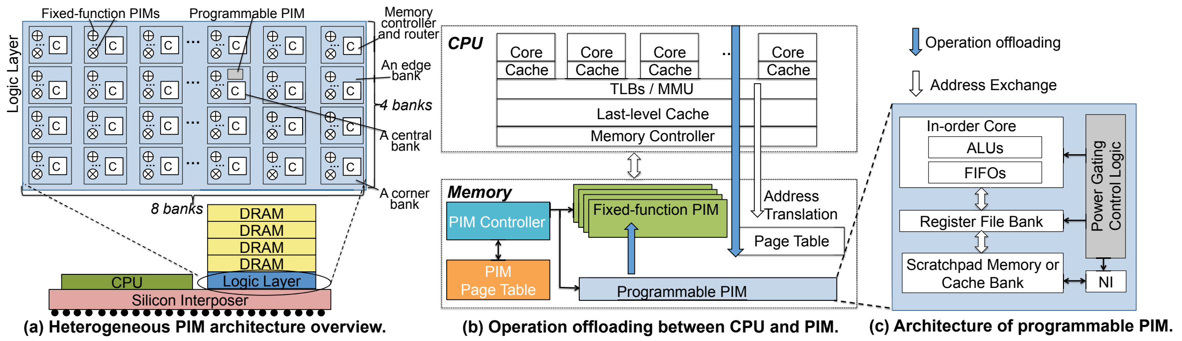


Fig. 3: Architecture overview of the proposed heterogeneous PIM.

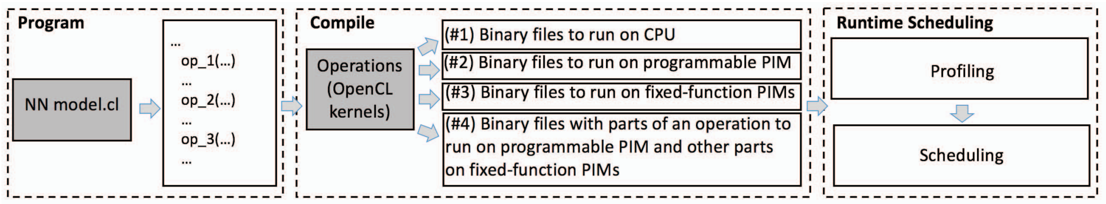

Program
NN model.cl
...
op_1(...)
...
op_2(...)
...
op_3(...)
...
Compile
Operations (OpenCL kernels)
(#1) Binary files to run on CPU
(#2) Binary files to run on programmable PIM
(#3) Binary files to run on fixed-function PIMs
(#4) Binary files with parts of an operation to run on programmable PIM and other parts on fixed-function PIMs
Runtime Scheduling
Profiling
Scheduling

Fig. 4: The process of executing NN training with our software framework design.

compute units, each of which is further divided into one or more processing elements (PE). An OpenCL program consists of kernels for compute devices and a host program. The host program runs on CPU and enqueues commands to a command-queue attached to a compute device.

In order to employ OpenCL programming model on the heterogeneous PIM system, we investigate how to map the heterogeneous PIM system onto the OpenCL model, and extend the OpenCL model for efficient runtime scheduling. In the following, we discuss our mapping method from the perspectives of platform model, execution model, and memory model. Table II summarizes our programming model extension.

Heterogeneous PIM platform model. Figure 5(b) illustrates our platform model. A large number of fixed-function PIMs provide massive parallelism for data processing. Each fixed-function PIM is a PE (in the OpenCL jargon). All fixed-function PIMs in all memory banks form a compute device. All fixed-function PIMs in a bank form a compute unit. Each programmable PIM is a compute device; each core of the programmable PIM is a PE. Hence, within the context of OpenCL, a heterogeneous PIM system has heterogeneous compute devices. We expose fixed-function PIM and programmable PIM as distinct compute devices to give control flexibility to the runtime system for operation scheduling. An OpenCL operation can be offloaded to any compute device that supports the operation execution.

Execution model. Tasks (i.e., operations in NN model training) to be launched on any PIM are represented as kernels managed by a host program, as in a traditional OpenCL program. If the task includes instructions that cannot be executed on the fixed-function PIM, then the task will not be scheduled by the OpenCL runtime to run on the fixed-function PIM. Otherwise, a task can run anywhere (CPU, fixed-function PIM, and programmable PIM). The OpenCL runtime (on CPU)

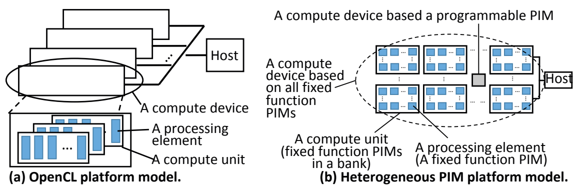

Host
A compute device
A processing element
A compute unit
(a) OpenCL platform model.
A compute device based a programmable PIM
A compute device based on all fixed function PIMs
Host
A compute unit (fixed function PIMs in a bank)
A processing element (A fixed function PIM)
(b) Heterogeneous PIM platform model.

Fig. 5: Enabling OpenCL platform model on heterogeneous PIM systems.

is in charge of task scheduling between different PIMs and CPU. Leveraging low-level APIs (Section IV-A) and hardware registers, the runtime can determine whether a specific PIM is busy and whether a specific task is completed. We describe the scheduling algorithm in Section III-C. Binary files for a task to run on CPU, fixed-function PIM, or programmable PIM are generated during the compilation stage. Given an OpenCL kernel for a task, we generate four binary files as shown in Figure 4. Section IV discusses details of binary generation.

Binaries (#3) and (#4) in Figure 4 allow *recursive PIM kernel*, a new execution scheme for our heterogeneous PIM design. A kernel in the programmable PIM can trigger data processing with fixed-function PIMs. This is supported by the programmable PIM runtime and implemented by calling small kernels loadable on fixed-function PIMs. By combining multiple kernels into a single kernel, the recursive PIM kernel scheme reduces overhead of kernel spawning and synchronization between the host and PIMs. Figure 6 shows an example that further explains the recursive PIM kernel. In the example, we illustrate an NN operation, Conv2DBackpropFilter, which is offloaded to the programmable PIM as a kernel; the kernel includes computation phases 1 and 2 that cannot be offloaded to the fixed-function PIMs. Conv2DBackpropFilter includes convolution computation ("Conv(...)" in the figure); The programmable PIM offloads this portion of computation to fixed-function PIM as a smaller kernel. The computation

TABLE II: Extending OpenCL for the heterogeneous PIM.

	Native OpenCL	Extensions for Heterogeneous PIM
Platform model	Host + accelerators (e.g., host + GPU).	Host + two types of accelerators (fixed-function PIMs and programmable PIM) driven by the characteristics of NN training.
Execution model	Host submits work to accelerators.	• Host submits work to accelerators; • Accelerators submit work to accelerators (i.e., recursive kernel invocation); • Work execution pipeline (i.e., operation pipeline); • Work scheduling based on dynamic profiling.
Memory model	• Multiple types of memory with a relaxed consistency model; • The global memory is not shared; • No defined synchronization across accelerators.	• A single global memory with a relaxed consistency model; • The global memory is shared; • Explicit synchronization across PIMs and CPU.

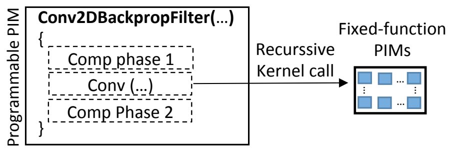


Fig. 6: An example of the recursive PIM kernel.

phases 1, 2 and convolution are combined as a single recursive PIM kernel, which reduces the synchronization between CPU and PIMs.

In general, the four binary files provide convenience for scheduling on CPU, the fixed-function PIMs and programmable PIM, and hence allows the runtime to maximize utilization of CPU and PIMs.

Memory model. The existing OpenCL defines four distinct memory regions in a compute device: global, constant, local, and private. On a heterogeneous PIM system, only a single global memory (i.e., the main memory) exists. In addition, the global memory is shared between CPU and PIMs, and addressed within a unified physical address space. This memory model requires synchronization at multiple points: (1) between CPU and PIMs; and (2) between different PIMs. The synchronization is necessary to avoid data race and schedule operations.

To implement effective synchronization, we employ the programmable PIM to drive the synchronization and avoid frequent interrupts to CPU. In particular, for synchronization between CPU and PIMs, the programmable PIM checks the completion of operations offloaded to PIMs (either programmable or fix function PIMs) and sends the completion information to CPU. For synchronization between different PIMs, the programmable and fix function PIMs synchronize through global variables in main memory.

Between CPU and PIMs, we introduce explicit synchronization points to synchronize the accesses to shared variables. To the host processor, the whole set of fixed-function PIMs or the programmable PIM appear as another processor. We employ standard synchronization schemes (e.g., barriers and locks), similar to the ones in a shared-memory multiprocessor. For fixed-function PIMs, their operations are atomic and the synchronization points are not expected in the middle of operations. For programmable PIMs, the synchronization points can be in the middle of a kernel. This is feasible based on global lock variables shared between CPU and PIMs. To support memory consistency, we adopt a relaxed memory consistency model, which aims to improve performance and reduce hardware complexity. In particular, an update to a memory location by a fixed-function PIM is not visible to all the other fixed-function PIMs at the same time. Instead, the local view of memory from each fixed-function PIM is only guaranteed to be consistent right after the kernel call to fixed-function PIMs. Between the fixed-function PIMs and programmable PIM, we employ the same consistency scheme: updates to memory locations by the entire set of fixed-function PIMs are not visible until the end of the kernel call to the fixed-function PIMs.

Because of our shared memory model, there is no data copy overhead before and after PIM kernel calls. Based on the above synchronization schemes, PIM kernel calls can be launched asynchronously to overlap computation on CPU and PIMs.

Support for easy program maintenance. To use the extended OpenCL programming model, operations need to be re-written using OpenCL. To write OpenCL code for operations, one can use OpenACC directives and compilers [19, 20] to automatically transform the original code into OpenCL code. This can significantly simplify the programming work. Furthermore, the number of operations for machine learning models is limited (tens of operations). Hence, using OpenCL to implement those machine learning operations is feasible. Other than that, however, the higher level software components (e.g., most of the middleware components, operation APIs, and Python syntax for using machine learning models) remain the same. This enables easy maintenance of machine learning frameworks.

C. Runtime System Design

Our runtime system is in charge of scheduling operations to fixed-function PIMs, programmable PIM, and CPU. To minimize NN training time, the runtime strives to maximize utilization of PIMs and CPU to optimize system throughput. The runtime schedules operations based on the following two steps.

Step 1: profiling. The runtime profiles performance of all operations on CPU. The profiling happens in only one step of NN model training. NN model training typically has a large amount of iterative steps (thousands and even millions of steps). Using one step for profiling has ignorable impact on performance. In addition, all steps almost have the same classes of operations; performance of operations (particularly execution time and the number of main memory access) remains stable across steps. Therefore, one step is sufficient for profiling purpose. During profiling, the runtime executes operations one by one in CPU, collecting execution time and the number of main memory access level cache misses of each operation with hardware counters. Based on the profiling results in the step, the runtime employs the following algorithm to determine the candidate operations to be offloaded to PIMs.

To determine the candidate operations, the runtime sorts operations into two lists (in descending order) based on execution time and the number of main memory accesses, respectively. Each operation in each of the two lists is correlated to an index, i.e., each operation has two indexes. With each operation, the runtime calculates a global index by adding these two indexes. Based on the global indexes, the runtime sorts operations into a global list. The runtime chooses top operations in the global list to offload to PIMs. Those top operations account for x% of total execution time of one step (x = 90 in our evaluation). The above algorithm is inspired by feature selection process in machine learning [22]. The goal of this algorithm is to select those operations that are both time-consuming and have a large number of main memory accesses.

Step 2: scheduling. Given the candidate operations to offload, the runtime makes the scheduling decision based on the following three principles.

- Scheduling operations to execute on fixed-function PIMs as much as possible.
- Scheduling operations to execute on PIMs (not CPU) as much as possible. In case all fixed-function or programmable PIMs are busy, the runtime will schedule the candidate operations to execute on CPU;
- Scheduling needs to respect data dependency across operations.

The first principle favors fixed-function PIMs over other compute units, because fixed-function PIMs are more energy efficient and typically performs faster with higher parallelism than other compute units. The second principle avoids CPU idling and introduces parallelism between CPU and PIMs. The third principle ensures execution correctness. Each operation defined in the machine learning frameworks typically has explicit input and output data objects (e.g., Tensors in TensorFlow), which provides convenience in tracking data dependencies across operations.

Operation pipeline. The above scheduling algorithm and principles enable operation pipeline to maximize hardware utilization. In particular, when an operation in a step cannot fully utilize fixed-function PIMs, our runtime schedules an operation in the next step to execute a portion of its computation by utilizing idling fixed-function PIMs as long as the two operations do not depend on each other.

In essence, these two operations can enable a pipelined execution manner. For instance, in AlexNet, a single convolution operation with a filter size of 11×11 consumes 121 multiplication and 120 addition (241 fixed-function PIMs in total). In case we have 444 fixed-function PIMs in total (Section IV-D), the utilization of fixed-function PIMs is only 54%. To improve hardware utilization, the runtime can schedule multiplication and addition from an operation (or operations) in the next step to execute on fixed-function PIMs. Once the convolution operation in the current step is completed, the partially executed operation(s) from the next step can immediately utilize the newly released fixed-function PIMs to improve hardware utilization and performance. This indicates that an operation can dynamically change its usage of PIMs, depending on the availability of PIMs. Such dynamic nature of operation execution is feasible based on a runtime system running on the programmable PIM (Section IV-C presents implementation details).

IV. Implementation

A. Low-level APIs for PIM Runtime System

We introduce several low-level API functions for fixed-function and programmable PIMs. These API functions allow direct control of individual PIMs, and provide foundation for our runtime. The API achieves the following functionality: (1) offloading a specific operation into specific PIM(s); (2) tracking the status of PIMs, including examining whether a PIM is busy or not; (3) querying the completion of a specific operation; (4) querying the computation location (i.e., which PIM) and input/output data location (i.e, which DRAM banks) for a specific operation. Table III summarizes our API functions.

B. OpenCL Binary Generation

To schedule operations to execute on CPU, fixed-function PIMs, or programmable PIM, we generate four binary files (Figure 4). In order to generate the binary file (#3) that corresponds to a portion of a large operation (an OpenCL kernel) to execute on fixed-function PIMs (e.g., the convolution within the operation Conv2DBackpropFilter), we first extract code sections from the corresponding OpenCL kernel. We then transform these code sections into a set of small kernels to execute on fixed-function PIMs. Finally we compile them into binary file (#3). In the original OpenCL kernel, these extracted code regions are replaced with the kernel calls and then compiled into binary file (#4) to execute on the programmable PIM. Binary files (#1) and (#2) are generated during the regular compilation stage.

C. Runtime Implementation

Our runtime consists of two components, which execute on the CPU and the programmable PIM, respectively.

The runtime on CPU. To support our runtime scheduling, we extend the runtime system of TensorFlow by adding approximately 2000 lines of code. The runtime on CPU schedules operations on CPU and PIMs, based on hardware utilization information provided by the low-level APIs. It does not support the implementation of recursive PIM kernels. In

TABLE III: Low-level APIs for PIMs.

Name	Description
int pim_fix(int* pim_ids, void* args, void* ret, size_t num_pim)	Asks specific fixed-function PIMs to work with input arguments `args` and return results `ret` and a work ID.
int pim_prog(int pim_id, pim_program kernel, void* args, int* args_offset, void* ret, size_t ret_size)	Asks a programmable PIM to work on a kernel (an operation) and return a work ID.
int pim_status(int pim_id)	Checks whether a specific PIM is busy.
int work_query(int work_id)	Checks whether a specific operation is completed.
void work_info(int work_id, int* pim_ids, int* data_loc)	Queries the computation location (pim_ids) and input/output data location (i.e, which DRAM banks) for a specific operation.

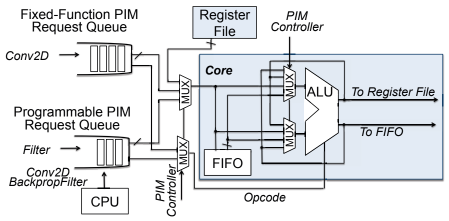


Fig. 7: Heterogeneous PIM implementation.

other words, the runtime on CPU is only responsible for offloading a kernel – which can have a part of its computation offloadable to fixed-function PIMs – to the programmable PIM. Our modifications to TensorFlow runtime include (1) device initialization and characterization using OpenCL intrinsics; (2) creating a device context and instance for a PIM device; (3) providing a new OpenCL device abstraction to other components of Tensorflow; (4) a mechanism to communicate with the runtime on the programmable PIM. This is one-time modification to Tensorflow, but can support various PIM hardware configurations without involving system programmers' future efforts.

The runtime on programmable PIM. The runtime on the programmable PIM supports recursive PIM kernels and operation pipeline. In particular, a kernel with a part of its computation replaced with kernel calls to fixed-function PIMs is handled by the runtime on the programmable PIM, which automatically offloads the computation to fixed-function PIMs. In order to keep track of the dynamic utilization of fixed-function PIMs, our runtime on the programmable PIM records the numbers of additions and multiplications already completed in each operation offloaded to the programmable PIM, as well as the remaining additions and multiplications.

D. Hardware Implementation

Figure 3 and Figure 7 illustrate our hardware implementation. The programmable PIM employs an ARM Cortex-A9 processor with four 2GHz cores. Each core has an in-order pipeline. In individual NN training models, operations that are potentially offloaded to the programmable PIM (e.g., ApplyAdam, MaxPooling, and ReLU) are typically not executed at the same time. Therefore, we only adopt one programmable PIM in our design. Even if we simultaneously train multiple NN models, the chance of having multiple operations to use the programmable PIM at the same time is low according to our evaluation with mixed workload analysis (Section VI-F).

Because a significant portion of NN training operations can be decomposed to addition and multiplications (Section II-A), we implement our fixed-function PIMs as 32-bit floating point multipliers and adders. We implement equal numbers of multipliers and adders in the pool of fixed-function PIMs. Our low-level APIs allow us to map operations to fixed-function PIMs that are in the same bank as input data of the operations. In addition, we accommodate random memory access pattern in NN computation by adopting buffering mechanisms [5]. We determine the fixed-function PIM configurations by employing a set of architectural level area, power, and thermal modeling tools, including McPAT [23] and HotSpot [24], to perform design space exploration of the logic die of 3D DRAM. Based on our study, the total number of allowed fixed-function PIMs is limited by the area of the logic die. With our baseline 3D DRAM configuration (Section V), we can distribute 444 fixed-function PIMs (pairs of multipliers and adders) across the 32 banks in the logic die. It is impossible to distribute these fixed-function PIMs evenly to each bank. We consider the placement of the fixed-function PIMs on 32 banks based on the following policy: we place more fixed-function PIMs on edge and corner banks than on central banks (Figure 3 (a)). The rationale behind is that the banks at the edge and corner have better thermal dissipation paths than central banks. Therefore, these banks can support higher computation density.

Furthermore, we employ a set of registers as shown in Figure 7. Each register indicates the idling of either a bank of fixed-function PIMs or the programmable PIM. The registers allow our software runtime scheduler to query the completion of any computation and decide the idleness of processing units.

V. Experimental Setup

A. Simulation Framework

In order to evaluate the performance of our design, we model fixed-function PIM and programmable PIM architectures, respectively, using Synopsys Design Compiler [25] and PrimeTime [26] with Verilog HDL. We adopt HMC 2.0 [27] timing parameters and configurations for our evaluation of 3D memory stack. Baseline memory frequency is set to 312.5 MHz, which is the same as HMC 2.0 specification [27]. This is also used as the working frequency of our heterogeneous PIM. We employ a trace generator developed on Pin [28] to collect instruction trace, when running our OpenCL kernel binaries on CPU. We develop a python-based, trace-driven simulation framework based on our design to evaluate the execution time of various training workload traces. Our simulator also incorporates our runtime scheduling mechanisms.

B. Power and Area Modeling

We adopt 10nm technology node for the host CPU and the logic die of the PIMs; 25nm technology node for the DRAM dies. We measure CPU and GPU power with VTune [29] and *nvidia-smi*, respectively. Our power model considers whole system power when we evaluate the power of heterogeneous-PIM-based systems, including CPU and the memory stack. We calculate the power and area of the programmable PIM

TABLE IV: System configurations.

CPU	Intel Xeon E5-2630 V3@2.4GHz
Main memory	16GB DDR4
Operating system	Ubuntu 16.04.2
GPU	NVIDIA GeForce GTX 1080 Ti (Pascal)
GPU cores	28 SMs, 128 CUDA cores per SM, 1.5GHz
L1 cache	24KB per SM
L2 cache	4096KB
Memory interface	8 memory controllers, 352-bit bus width
GPU main memory	11GB GDDR5X

using McPAT [23]. We evaluate the power and area of fixed-function PIMs using Synopsys Design Compiler [25] and PrimeTime [26].

C. Workloads

We evaluate various training models, including VGG-19 [1], AlexNet [2], Deep Convolutional Generative Adversarial Networks (DCGAN)) [12], ResNet-50 [30], Inception-v3 [31], Long Short Term Memory (LSTM) with dropout [32] and Word2vec [33]. LSTM and Word2vec are evaluated in Section VI-F. The rest models are widely used in recent studies on CNN training and image classification.

Training Datasets. We employ ImageNet as training data set of VGG-19, AlexNet, ResNet-50, and Inception-V3. ImageNet is a large image dataset with millions of images belonging to thousands of categories. DCGAN employs MNIST dataset [34]. LSTM adopts Penn Tree Bank (PTB) [32] dataset. Word2vec employs "questions-words" dataset [35] in TensorFlow.

Training framework and batch Size. We adopt TensorFlow [7] as our training framework. We adopt default batch sizes of each training model in TensorFlow. The batch size of VGG-19, AlexNet and Inception-v3 is 32. The batch size of Word2vec and ResNet-50 is 128. DCGAN has a batch size of 64. LSTM employs a batch size of 20.

D. Real Machine Configurations

To compare performance and energy efficiency of heterogeneous PIM with GPU and CPU, we run the training models on (1) NVIDIA GeForce GTX 1080 Ti graphic card [36] and (2) CPU listed in Table IV. Our GPU-based training evaluations adopt CUDA 8 [37] and NVIDIA cuDNN 6.0 library [38]. GPU utilizations of each training model in TensorFlow are: Inception-v3 (average: 62%; peak: 100%); ResNet-50 (average: 44%; peak: 58%); AlexNet (average: 30%; peak: 34%); VGG-19 (average: 63%; peak: 84%); DCGAN (average: 28%; peak 42%. We use NVIDIA's profiling tool [39] and Intel's VTune [14] to collect performance and power statistics.

VI. Evaluation

Our experiments compare among the following five configurations, including our design.

- **CPU** – Executing all training operations on CPU;
- **GPU** – Executing all training operations on GPU;
- **Progr PIM** – Programmable PIMs only, which executes all operations on as many ARM-based programmable cores as needed by workloads (without our runtime scheduling);
- **Fixed PIM** – Fixed-function PIMs only, which executes the operations that can be offloaded on fixed-function PIM and other operations on CPU (without our runtime scheduling);
- **Hetero PIM** – Our heterogeneous PIM design (including our runtime scheduling).

A. Execution Time Analysis

Figure 8 shows execution (training) time of various NN training models. We break down the execution time into synchronization time, data movement time and operation time (i.e., computation time in CPU, GPU or PIMs). For GPU-based systems, the data movement time is the time for data transfer between main memory and GPU global memory. Certain amount of data transfer time is overlapped with GPU computation, e.g. by copying a minibatch of images to the GPU memory, while the computation on GPU is processing another minibatch. Our breakdown only shows the data transfer time that is not hidden by the computation. For PIM-based systems, the data movement time is the time for data transfer between CPU and the main memory. Our runtime scheduling allows operations to execute concurrently on CPU and PIMs.

We observe that PIM-based designs (including Fixed PIM, Progr PIM and Hetero PIM) perform much better than CPU, with 19%-28× performance improvement. Compared with Progr PIM and Fixed PIM, our design has 2.5×-23× and 1.4×-5.7× performance improvement, respectively. PIM-based designs also significantly reduce data movement overhead, compared to CPU and GPU. Overall, Hetero PIM leads to the lowest synchronization and data movement overhead among all configurations.

The performance benefit of Hetero PIM stands out with larger training models and larger working sets due to (i) more reduction in data movement and (ii) higher parallelism between host CPU and PIMs introduced by more offloadable operations. DCGAN has smaller model and working set than others. Therefore, Hetero PIM appears to result in worse performance than GPU with DCGAN; yet, compared with other configurations, our design still significantly improves performance. ResNet is a large training model with large working sets. As a result, Hetero PIM leads to better performance than GPU with ResNet. With other training models, Hetero PIM leads to performance close to (within 10% of) GPU. GPU has good performance because of its massive thread-level parallelism. Our design leads to much better performance than all other configurations.

B. Energy Consumption Analysis

Figure 9 shows the dynamic energy consumption of the five NN models with five different configurations. The energy consumption results are normalized to the results of Hetero PIM. We observe substantial energy benefit of using our design: it consumes 3×-24× and 1.3×-5× less energy than CPU and GPU, respectively. CPU consumes higher dynamic energy than Hetero PIM, Fixed PIMs, and GPU, even though its power consumption is the lowest among all of these configurations (note that we take CPU power into account when we calculate the power of PIMs and GPU, in order to evaluate full-system

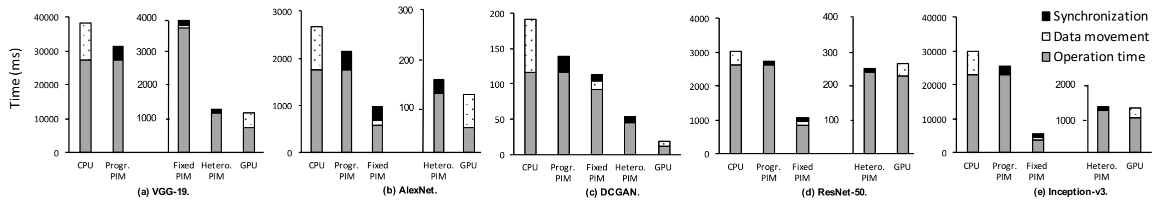


Fig. 8: Execution time breakdown of five NN models.

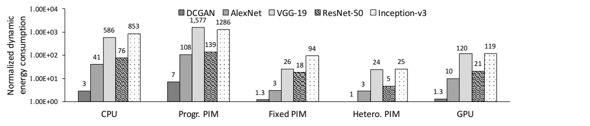


Fig. 9: Normalized dynamic energy of various NN models.

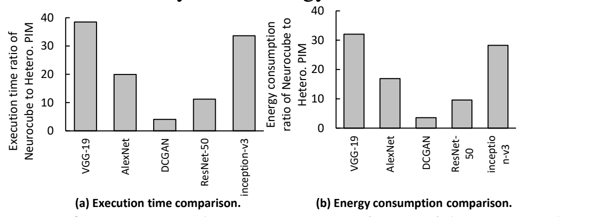


Fig. 10: Performance and energy comparison with Neurocube.

power consumption). This is because CPU has the longest execution (training) time. Furthermore, we notice that the dynamic energy consumption of Progr PIM is higher than that of other configurations, because the speed of Progr PIM is only slightly faster than that of CPU, yet the dynamic power of Progr PIM is higher than that of CPU due to the additional processing units in Progr PIM. Overall, Hetero PIM leads to the lowest dynamic energy consumption across all configurations.

C. Comparison with Prior PIM-based NN Acceleration

Figure 10 shows a quantitative comparison between our design and a recent PIM-based NN accelerator design, Neurocube [6] (qualitative comparison is in Section VII). Neurocube also reduces data movement overhead and improves energy efficiency by using PIM technology. However, our work outperforms Neurocube in terms of performance and energy efficiency. With highly compute-intensive models, such as VGG-19 and Inception-V3, our design achieves much higher performance and energy-efficiency improvement than Neurocube. Even with less compute-intensive models, such as DCGAN, our work can achieve at least 3× higher performance and energy efficiency than Neurocube. The reason for the improvement is two-fold: (1) Neurocube only adopts programmable PIMs, while our design employs energy-efficient, highly-parallel fixed-function PIMs to accelerate fine-grained operations; (2) Our design employs runtime scheduling that effectively optimizes hardware utilization (evaluated in Section VI-E).

D. Sensitivity Study

Frequency Scaling. We adopt three different frequencies for fixed-function PIMs and programmable PIM: their original frequencies (1×), doubling of their frequencies (2×) and quadrupling of their frequencies (4×). We use a phase-locked loop module to change the frequency. We study execution (training) time with the different frequencies.

Figure 11 shows the results. We observe that with higher frequency, the heterogeneous PIM performs better than GPU. With 2× frequency, Hetero PIM performs 36% and 17% better than GPU, with VGG-19 and AlexNet, respectively. With 4× frequency, Hetero PIM performs 37% and 60% better than GPU, with VGG-19 and AlexNet respectively. We also observe that the synchronization and data movement overheads are reduced, when using higher frequencies.

Programmable PIM Scaling. We employ three different configurations for Hetero PIM, while keeping the area of logic die in the memory stack unchanged. We scale the number of Progr PIM (ARM cores) from one to two to 16, while the rest of the logic die area is used to implement Fixed PIM. The three configurations are labeled as *1P*, *4P* and *16P*, respectively.

Figure 12 shows our results. The figure reveals that the performance difference between the three configurations is relatively small. The performance difference between *16P* and *1P* is 12%–14%. The reason is two-fold: (1) One Progr PIM is sufficient for the NN models to schedule and pipeline operations; (2) Using more Progr PIMs loses more Fixed PIMs, given the constant area in the logic layer of memory stacks.

E. Evaluation of Software Impact

We isolate the the impact of our software (runtime) techniques from that of Hetero PIM hardware. We aim to provide more insightful analysis on the effectiveness of software/hardware co-design. In particular, we study execution time, energy and utilization of Fixed PIM with and without the recursive PIM kernel call (RC) and operation pipeline (OP) – our two major runtime techniques. *Without RC and OP, we also compare Hetero PIM hardware design with Fixed PIM and Progr PIM, in terms of execution time and energy. This comparison allows us to study the impact of Hetero PIM architecture with the absence of our runtime techniques.*

Execution time analysis. As shown in Figure 13, Hetero PIM without runtime scheduling performs better than Progr PIM and Fixed PIM by up to 8.5×. This demonstrates the necessity of using Hetero PIM architecture. However, comparing with Fixed PIM, the performance benefit of Hetero PIM hardware is not significant (7%-30%). After incorporating the runtime scheduling techniques, the performance of Hetero PIM is improved by up to 3.8×. This result demonstrates the necessity of using an efficient runtime to maximize the benefit of Hetero PIM architecture.

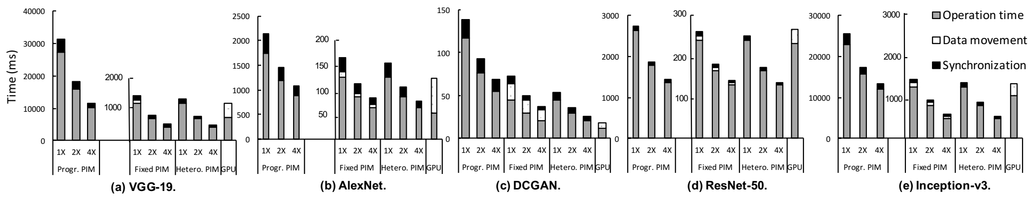


Fig. 11: Execution time breakdown of various NN models with 3D memory frequency scaling.

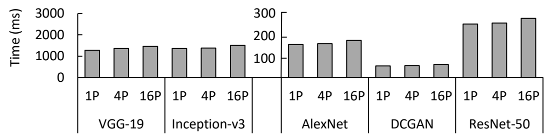


Fig. 12: Execution time with Progr PIM scaling.

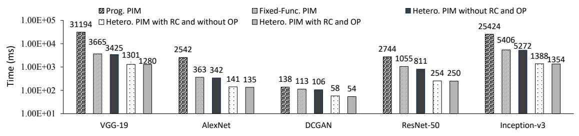


Fig. 13: Execution time with and without RC and OP.

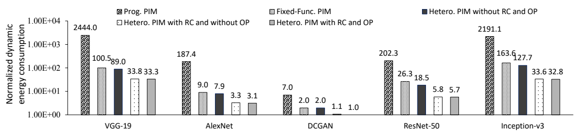


Fig. 14: Dynamic energy with and without RC and OP.

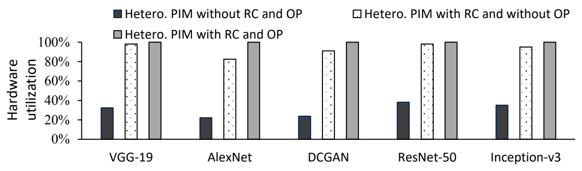


Fig. 15: Hardware utilization with and without RC and OP.

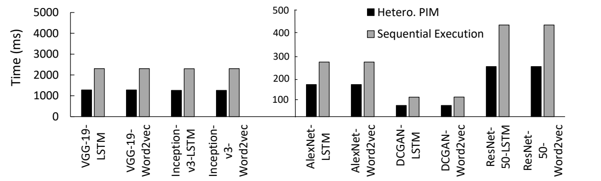


Fig. 16: Execution time of multiple NN training models with our design and sequential execution, respectively.

Energy analysis. Figure 14 shows our energy results normalized to the energy of Hetero PIM with RC and OP. We have similar observations as the execution time analysis: Hetero PIM without runtime scheduling performs better than Progr PIM and Fixed PIM by up to 2.7×. With RC and OP, we further reduce the energy of Hetero PIM by up to 3.9×.

PIM utilization analysis. Figure 15 shows our utilization results. With RC only, the utilization of Fixed PIM in Hetero PIM is improved by up to 66% (VGG-19); With OP, the utilization of Fixed PIM is further improved by up to 18% (AlexNet); With RC and OP, the utilization of Fixed PIM is close to 100%. The reason for the poor hardware utilization with neither RC nor OP is the lack of scheduling for the operations that do not have sufficient parallelism or cannot be completely offloaded to Fixed PIM.

F. Mixed Workloads Analysis

We also evaluate the case, when multiple models co-run in the same system [40]. We co-run two NN training models: a CNN model and a non-CNN model. The CNN model can execute on CPU and PIMs, subject to our runtime scheduling; the non-CNN model executes on CPU or the programmable PIM, when they are idle. Figure 16 shows the results of six co-run cases. In each case, "Hetero. PIM" indicates that we simultaneously execute both models, with the total execution time matched (i.e., when the CNN model executes for one step, the non-CNN model can execute one or multiple steps because the latency of the CNN model in one step can be longer than the non-CNN model in one step); "Sequential Execution" indicates that we execute the two models one after another in serial.

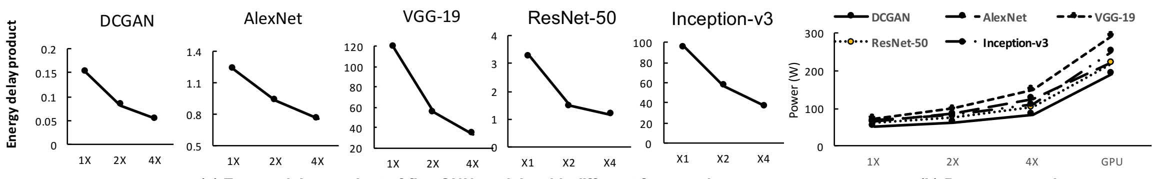
The results show that Hetero. PIM achieves 69%-83% performance improvement compared with Sequential Execution. Such improvement comes from high utilization of CPU and the programmable PIM in our design. With Sequential Execution, there can be no operations available to execute even though CPU and the programmable PIM are idle due to dependency between operations within the same model. Hetero. PIM avoids hardware idling, because operations across different models have no dependency and can execute simultaneously.

G. Energy Efficiency Analysis

We study energy efficiency of the PIMs with different frequencies as in SectionVI-D. We use energy-delay-product (EDP) as the metric to evaluate energy efficiency. Figure 17 (a) shows the results. The figure reveals that the most energy efficient point is not the original frequency for the five models. The 4× frequency is the most energy efficient for the five models. The tradeoff between energy consumption and execution time leads to such results. Thus, we conclude that higher frequency tends to be more energy efficient for NN model training. Figure 17 (b) compares power consumption between GPU and Hetero PIM with different frequencies. In general, GPU is very power hungry. It consumes 1.5× to 2.6× more power than Hetero PIM with high frequency (4×). Compared with GPU, Hetero PIM can be highly power efficient.

VII. Related Work

To our knowledge, this is the first paper to propose a software/hardware co-design of a heterogeneous-PIM-based acceleration framework for NN training. Whereas previous PIM-based accelerator designs [5, 9, 10, 41–45] investigated



(a) Energy delay product of five CNN models with different frequencies. **(b) Power comparison.**

Fig. 17: Energy efficiency and power with 3D memory frequency scaling.

the mapping of workloads on either fixed-function or programmable PIMs, it is unclear how to coordinate software and hardware designs to best utilize PIM technologies to support the heterogeneity requirement of NN training workloads.

Processing-in-memory for machine learning. Recent PIM-based machine learning accelerator designs strive to leverage the memory cells of nonvolatile memory technologies to execute NN inference operations [5, 46–48]. However, NN training typically incorporates substantial complex operations as we identified. It is difficult to accommodate these complex operations in previous processing-in-memory-cell designs. Azarkhish et al. [49] and Schuiki et al. [50] adopt RISC-V cores [51] and a streaming coprocessor in die-stacked DRAM to accelerate convolution networks or SGD. However, the RISC-V cores are merely used to control the arithmetic elements in the streaming coprocessor. Furthermore, both designs require users to modify code and perform tiling based on new APIs. Schuiki et al.'s study [50] only focuses on a specific operation (SGD). Azarkhish et al.'s design [49] primarily aims at inference and requires data to be carefully laid out in memory with 4D tiling. This constraint on data layout leads to inefficient training, because intermediate activations after each layer need to be re-tiled [50]. Neurocube [6] accelerates CNN inference and training by integrating programmable processing elements in the logic layer of 3D die-stacked DRAM. However, using programmable PIMs alone cannot provide the massive parallelism and execution efficiency enabled by heterogeneous PIMs. Furthermore, the aforementioned previous studies do not consider dynamic runtime scheduling of operations. Our experiment results demonstrate an efficient heterogeneous PIM design with runtime scheduling.

Processing-in-memory for general applications. Fujiki et al. [9] proposed a ReRAM-based in-memory processor architecture and data-parallel programming framework. The study introduces a compact instruction set for memory array with processors. The programming framework combines dataflow and vector processing, employs TensorFlow input, and generates code for in-memory processors. Our work also employs TensorFlow, but optimizes operations scheduling and introduces PIM heterogeneity. Ahn et al. [41] explores mapping of PIM operations based on data locality of applications, while we schedule operations in multiple dimensions – hardware utilization, data locality, and data dependency. Ahn et al. [45] introduced PIM for parallel graph processing. The design offers an efficient communication method between memory partitions and develops prefetchers customized for memory access patterns of graph processing. Other works introduce PIM architectures based on 3D-stacked memory. For example, Zhang et al. [52] presented an architecture for programmable, GPU-accelerated, in-memory processing implemented using 3D die-stacking. The throughput-oriented nature of GPU architectures allows efficient utilizaztion of high memory bandwidth provided by 3D-stacked memory, while offering the programmability required to support a broad range of applications. Akin et al. [42] presented a set of mechanisms that enable efficient data reorganization in memory using 3D-stacked DRAM. However, the aforementioned studies cannot efficiently accelerate NN training workloads, because they cannot fully accommodate the heterogeneous computing requirement in NN training. Furthermore, these studies do not consider efficient programming model and runtime system to accommodate the hardware heterogeneity as explored in our study.

Other accelerator optimization for machine learning. Recent works explored software- and hardware-based approaches for a variety of inference acceleration [53–57]. Most of these works focused on improving performance and energy efficiency of NN inference. However, training is much more compute and memory intensive than inference. The data movement overhead in training is much more significant. Several prior studies [58–60] investigated architecture design for NN training. However, these studies focus on addressing the memory capacity constraint issues caused by a large amount of feature maps generated in CNN training. The data movement bottleneck is not fully explored.

VIII. Conclusions

In this paper, we propose a software and hardware co-design of heterogeneous PIM approach, combining the power of programmable PIM and fixed-function PIMs, for NN training. Our software design enables (1) a portable and unified programming model across CPU, fixed-function PIMs, and programmable PIM; (2) runtime scheduling that effectively optimizes PIM hardware utilization and maximizes NN-operation-level parallelism. Our design not only allows natively training models to execute on heterogeneous PIM, but also enables easy maintenance of machine learning frameworks. Our design achieves significant improvement in performance and energy efficiency with various NN training workloads.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This paper is supported in part by NSF grants 1652328, 1718158, 1617967, 1553645, 171819, and SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory.

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[3] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, "Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent," in *International Symposium on Computer Architecture (ISCA)*, 2017.

[4] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism," in *International Symposium on Computer Architecture (ISCA)*, 2017.

[5] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 27–39, 2016.

[6] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 380–392, 2016.

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[8] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[9] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), pp. 1–14, ACM, 2018.

[10] G. H. Loh, N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. Meswani, D. Zhang, and M. Ignatowski, "A processing-in-memory taxonomy and a case for studying fixed-function PIM," in *WoNDP*, pp. 1–6, 2013.

[11] "Khronos Group, the open standard for parallel programming of heterogeneous systems." https://www.khronos.org/opencl/.

[12] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *CoRR*, vol. abs/1511.06434, 2015.

[13] "TensorBoard: Visualizing learning," https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard.

[14] "Intel, Vtune user's guide," https://software.intel.com/en-us/get-started-with-vtune/.

[15] S. Boag, P. Dube, B. Herta, W. Hummer, V. Ishakian, J. K. R., M. Kalantar, V. Muthusamy, P. Nagpurkar, and F. Rosenberg, "Scalable Multi-Framework Multi-Tenant Lifecycle Management of Deep Learning Training Jobs," in *Workshop on ML Systems at NIPS'17*, 2017.

[16] "MultiModel: Multi-task machine learning across domains," https://ai.googleblog.com/2017/06/multimodel-multi-task-machine-learning.html.

[17] Y. Zhu, B. Wang, D. Li, and J. Zhao, "Integrated thermal analysis for processing in die-stacking memory," in *Proceedings of the Second International Symposium on Memory Systems*, pp. 402–414, 2016.

[18] "NVIDIA, TITAN Xp," https://www.nvidia.com/en-us/geforce/products/10series/titan-xp/.

[19] "Openarc." https://ft.ornl.gov/research/openarc.

[20] "Ipmacc compiler." https://github.com/lashgar/ipmacc.

[21] O. Forum, "OpenMP Fortran application program interface, version 1.1." http://www.openmp.org, 1999.

[22] H. v. Halteren, J. Zavrel, and W. Daelemans, "Improving accuracy in word class tagging through the combination of machine learning systems," *Computational linguistics*, vol. 27, no. 2, pp. 199–229, 2001.

[23] 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.

[24] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, "Temperature-aware microarchitecture: modeling and implementation," *ACM Transactions on Architecture and Code Optimization*, vol. 1, no. 1, pp. 94–125, 2004.

[25] Synopsys, "Design compiler." https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html.

[26] Synopsys, "Primetime." https://www.synopsys.com/support/training/signoff/primetime1-fcd.html.

[27] HMCC, "Hybrid memory cube specification 2.0." http://http://www.hybridmemorycube.org/.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, (New York, NY, USA), pp. 190–200, 2005.

[29] J. Reinders, "Vtune performance analyzer essentials," *Intel Press*, 2005.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[31] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for

computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818–2826, 2016.

[32] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[33] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[34] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[35] "Tensorflow, questions-words dataset," http://download.tensorflow.org/data/questions-words.txt.

[36] "NVIDIA, GeForce GTX 1080 Ti," https://www.nvidia.com/en-us/geforce/products/.

[37] "NVIDIA CUDA." http://www.nvidia.com/cuda.

[38] NVIDIA, "cudnn." https://developer.nvidia.com/cudnn.

[39] "NVIDIA, Profiler user's guide," http://docs.nvidia.com/cuda/profiler-users-guide/.

[40] T. Li, J. Zhong, J. Liu, W. Wu, and C. Zhang, "Ease.ml: Towards multi-tenant resource sharing for machine learning workloads," *Proc. VLDB Endow.*, vol. 11, no. 5, 2018.

[41] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pp. 336–348, 2015.

[42] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3D-stacked DRAM," in *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 131–143, 2015.

[43] L. Nai and H. Kim, "Instruction offloading with HMC 2.0 standard: A case study for graph traversals," in *Proceedings of the 2015 International Symposium on Memory Systems*, pp. 258–261, 2015.

[44] Y. Eckert, N. Jayasena, and G. H. Loh, "Thermal feasibility of die-stacked processing in memory," in *WoNDP*, pp. 1–6, 2014.

[45] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 105–117, 2015.

[46] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 14–26, 2016.

[47] P. Wang, Y. Ji, C. Hong, Y. Lyu, D. Wang, and Y. Xie, "SNrram: an efficient sparse neural network computation architecture based on resistive random-access memory," in *Proceedings of the 55th Annual Design Automation Conference*, p. 106, ACM, 2018.

[48] S. Angizi, Z. He, A. S. Rakin, and D. Fan, "CMP-PIM: an energy-efficient comparator-based processing-in-memory neural network accelerator," in *Proceedings of the 55th Annual Design Automation Conference*, pp. 105–110, ACM, 2018.

[49] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 420–434, 2018.

[50] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *arXiv*, vol. abs/1803.04783, 2018.

[51] "RISC-V: The free and open RISC instruction set architecture." https://riscv.org/.

[52] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, pp. 85–98, 2014.

[53] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 267–278, 2016.

[54] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, pp. 367–379, IEEE, 2016.

[55] H. Esmaeilzadeh, A. Sampson, and L. Ceze et al., "Neural acceleration for general-purpose approximate programs," in *MICRO*, pp. 449–460, IEEE Computer Society, 2012.

[56] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *MICRO*, pp. 1–12, IEEE, 2016.

[57] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *IEEE/ACM International Symposium on Microarchitecture*, 2014.

[58] M. Rhu, N. Gimelshein, and J. C. et al., "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *MICRO*, 2016.

[59] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *HPCA*, pp. 78–91, 2018.

[60] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *ISCA*, pp. 1–14, 2018.