AnalyzeThat: A Programmable Shared-Memory System for an Array of Processing-In-Memory Devices

Sangkuen Lee[1], Hyogi Sim[1], Youngjae Kim[2*], Sudharshan S. Vazhkudai[1*]
[1]Oak Ridge National Laboratory, [2]Sogang University
{lees4,simh,vazhkudaiss}@ornl.gov, youkim@sogang.ac.kr

Processing In Memory (PIM), the concept of integrating processing directly with memory, has been attracting a lot of attention since PIM can assist in overcoming the throughput limitation caused by data movement between CPU and memory. The challenge, however, is that it requires the programmers to have a deep understanding of the PIM architecture to maximize the benefits such as data locality and parallel thread execution on multiple PIM devices. In this study, we present AnalyzeThat, a programmable shared-memory system for parallel data processing with PIM devices. Thematic to AnalyzeThat is a rich PIM-Aware Data Structure (PADS), which is an encapsulation that integrally ties together the data, the analysis tasks and the runtime needed to interface with the PIM device array. The PADS abstraction provides (i) a key-value data container that allows programmers to easily store data on multiple PIMs, (ii) a suite of parallel operations with which users can easily implement data analysis applications, and (iii) a runtime, hidden to programmers, which provides the mechanisms needed to overlay both the data and the tasks on the PIM device array in an intelligent fashion, based on PIM-specific information collected from the hardware. We have developed a PIM emulation framework called AnalyzeThat. Our experimental evaluation with representative data analytics applications suggests that the proposed system can significantly reduce the PIM programming effort without losing its technology benefits.

I. Introduction

Processing-in-memory (PIM) is a well-known concept of integrating processing units (cores) with memory devices in order to reduce memory latency and increase memory bandwidth [1]. PIM was originally introduced more than a decade ago, with several studies showing its potential advantages in various applications such as knowledge discovery, scientific computing, image processing and databases [2], [3], [4], [5], [6]. However, due to the difficulty in the heterogeneous manufacturing process of logic and memory, so far, PIM has not been widely adopted in commodity systems [7].

Recently, there has been a renewed interest in PIM from academia and industry [8]. Emergence of advanced 3D memory allows the stacking of memory chips atop a processing unit (e.g., GPU), enabling processing near memory, e.g., TOP-PIM [9] that was put forward by AMD. Even without 3D memory, specialized PIM devices have been prototyped by Micron in their "Automata Processor" in November 2013, which is a DRAM chip with an array of processors [10]. NVRAM such as PRAM [11], memristors [12], and STT-MRAM [13] is expected to replace DRAM as it is non-volatile and power-efficient. PIM architectures can also employ such emerging memory in place of DRAM. However, since NVRAM is limited in write cycles, the write frequency on the memory needs to be carefully controlled.

Exploiting PIM architectures has potential advantages for processing big data in terms of both energy efficiency and processing time [7], [14]. Not only can multiple PIM cores process data in a parallel fashion, but each PIM core can also achieve higher data processing throughput than commodity systems by accessing data stored in its corresponding local memory device with lower latency [9], [15], [14]. The concept of PIM is now also acknowledged to be useful in extreme-scale systems, where power consumption is increasingly becoming a significant design constraint.

For example, the U.S. Department of Energy's (DOE) CORAL consortium is deploying three O(100) petaflops systems, SUMMIT, SEIRRA and AURORA systems at Oak Ridge National Laboratory (ORNL), Lawrence Livermore National Laboratory (LLNL) and Argonne National Laboratory (ANL), respectively in the 2018-2019 timeframe, which are expected to consume in the range of 10-13MW of power. These systems will be equipped with deep memory tiers ranging from tens of GBs of high-bandwidth memory (HBM), several PBs of DRAM and persistent memory. In such systems, DRAM is a significant source of power consumption. The DOE's future exascale system in 2023 is expected to be built within an energy envelope of 20MW. At exascale, it is widely expected that the cost of data movement between the deep memory tiers will rival the cost of computation itself [16]. While technologies such as HBM and persistent memory help alleviate this concern, PIM and processing near memory can be a significant step in this direction as well.

However, it is a significant challenge to integrate PIM architectures into extant user application software. Programming the PIM architecture can be a non-intuitive task for users, since it is necessary to properly distribute data and tasks to multiple PIM devices to fully take advantage of data locality and parallelism of the PIM devices [9]. If memory allocation and concurrency control are not accomplished

*Corresponding Author

properly, efficiency and scalability of the application can significantly decrease due to data skew [17] and massive remote access [18].

To address these challenges, we present AnalyzeThat, a programmable shared memory system for PIM devices. The system provides an abstraction for parallel data processing with PIM devices, which allows programmers to focus on the functionality of their programs and not on the management of data placement and thread concurrency. More specifically, the abstraction is provided as a PIM-Aware Data Structure (PADS), which is a collection of key-value pairs distributed over multiple PIM devices. Programmers load their data into the PADS objects and execute the workflow of data analytics applications with PADS parallel operations. Thereafter, AnalyzeThat's runtime is responsible for making decisions to efficiently process the PADS parallel operations on a system having an array of PIM devices. The decisions (e.g., how to distribute key-value pairs and which PIM to offload a task to) can be made based on PIM-specific information that is collected from hardware devices. For such a capability, AnalyzeThat exploits operating system and hardware support (e.g., device driver and global address space). Further, to give more control to programmers to achieve generality of PIM programming, a low-level PIM programming library that resembles POSIX pthread APIs is provided as part of the AnalyzeThat library suite.

II. AnalyzeThat Programmable System

Figure 1 depicts the interactions between various components of AnalyzeThat.

- **Array of PIM Devices** At the lowest level is an array of PIM devices, capable of processing. As we will discuss in § III-A, emerging hardware technologies support a single shared memory address space for a node composed of general CPUs and compute accelerators such as PIM, where any core can globally access any memory region [19], [20]. The PIM device can either be 3D stacked with memory chips layered atop the logic chip on the same die or a discreet PIM device with an embedded controller.
- **PIM Device Driver** We have developed a first order implementation of a device driver that each PIM device needs to support in order to realize the functionality needed in the AnalyzeThat system. In our implementation, the device driver is responsible for providing the communication path between the PIM hardware and the higher-level components in AnalyzeThat, maintaining PIM-specific internal information (e.g., wearout, data load) that will be queried for data placement, task offloading and data segment expansion.
- **Low-Level PIM Programming Library** Atop the PIM array and the device driver is a low-level PIM programming library, which gives direct control of PIM devices to programmers, such as allocating PIM memory and offloading a task to a PIM device. An advanced programmer can implement his application directly using the low-level library.

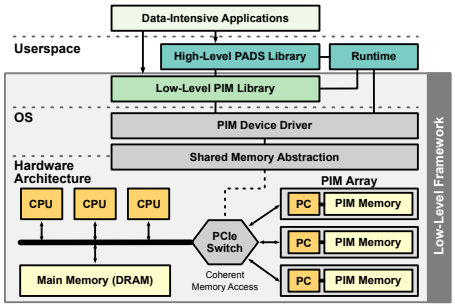


Figure 1: The hardware and software architecture of AnalyzeThat. PC denotes the PIM-core.

- **PADS (PIM-Aware Data Structure)** The PADS library provides an easier way of programming with PIM devices. PADS provides an object-oriented abstraction that tightly couples a data object and its operations. Specifically, PADS provides an abstraction of a *key-value container*, hiding hardware-specific details (e.g., a PIM location) from programmers. In addition, the PADS key-value container supports a *suite of parallel operations*, such as *map()* and *reduce()*, which can be used to manipulate the key-value dataset in parallel. For instance, a programmer can create a PADS object and populate it from an input file. The PADS library then transparently distributes the input data across multiple PIM devices based on a data placement policy. Thereafter, the programmer can perform data manipulation in parallel by invoking the PADS operations.
- **Runtime Environment** Behind the PADS abstraction, a runtime handles the hardware-specific details such as data distribution and task execution across the PIM array. For this, the runtime periodically collects the status of each PIM device using the low-level library and the device driver. Based on the device status, the runtime dynamically makes decisions on data and task load distribution.

AnalyzeThat utilizes the PIM cores to process the data in parallel. *Together, these constructs provide a very potent, programmable shared memory abstraction for in-situ data analytics atop an array of PIM devices.*

III. AnalyzeThat Low-Level Framework

A. Hardware Architecture

Each PIM device in AnalyzeThat is composed of a dedicated computing unit (PIM core), a set of programmable registers and memory chips (PIM memory). The PIM core is a *fully programmable* low-power processor similar to ARM processors [21]. This allows any general program to be executed on the devices and, therefore, grants more flexible programming than FPGA-based accelerators [22]. However, due to the difference in the instruction set architectures between the host CPU and the PIM core, the target binary code that is to be executed on the PIM core should be compiled and built separately with a supporting compiler. In addition to general programmability, each PIM core contains its own hardware cache and MMU (Memory Management Unit), and runs a firmware to control the internal hardware

Low-level	`void* pimmalloc(size_t size, int pim)`	Allocates `size` bytes of memory in PIM device `pim`
	`void pimfree(void* addr)`	Frees memory of address `addr`
	`int pimexec_exec(pimexec_data_t* pe)`	Initiate offloading of a user-defined function
	`int pimexec_wait(pimexec_data_t* pe)`	Block the current thread until the execution completes
PADS	`void PADS.import(char* file, parser_t* pf)`	Import data from a `file` using a parser function `pf`
	`void PADS.map(PADS& out, mapper_t* mf, void* arg)`	Performs a user-defined map function `mf` and stores results in `out`
	`void PADS.reduce(PADS& out, reducer_t* rf, void* arg)`	Performs a user-defined reduce function `rf` and stores results in `out`
	`void PADS.export(char* file)`	Export data into `file`

Table I: Example APIs of low-level PIM library in C and high-level PADS library in C++. `pimexec_data_t` is a record type that encapsulates all information regarding a code execution on a PIM core.

operations. The programmable registers can be read and written by host applications to initiate a task execution or to fetch runtime information, e.g., memory duty cycles. Such information is used by the AnalyzeThat runtime (§ V).

As shown in Figure 1, multiple PIM devices are connected to a single host via a fast switch interconnect, i.e., PCI Express. AnalyzeThat exploits the emerging memory interconnect protocol, i.e., Cache Coherent Interconnect for Accelerators (CCIX) [19], which allows cache coherent accesses across heterogeneous memory devices from different processors and accelerators. Note that CCIX protocol does not require any modifications to existing system software or operating systems, because the protocol is fully implemented in hardware and firmware [19]. The coherent memory access protocol and the fast switch interconnect allow host CPUs to directly access the PIM memories without having to explicitly transfer data between the host and the PIM memories. Similarly, a PIM core can access not only its own local PIM memory but also other remote PIM memories and DRAM on the host. Overall, this shared memory abstraction from the hardware enables us to project a consistent virtual address space to both the offloaded PIM task and its parent application on the host, e.g., a memory pointer can be shared between them, and greatly facilitates application development.

B. PIM Device Driver

In addition to its capability as a memory storage device, the PIM device features its own processing power. To exploit the processing power, e.g., execute a task using a PIM core, a host software needs to communicate with a PIM device, which requires access to the programmable registers of the PIM device. The PIM device driver primarily assists userspace programs to allocate memory space and also to access the programmable registers by providing a memory-mapped I/O interface. Specifically, during system initialization, the device driver detects available PIM devices that are connected to the host. For each PIM device, the device driver then creates a dedicated device file, i.e., `/dev/pimN`, and */proc* entries, i.e., `/proc/pim/pimN/` on the host. Also, a set of *ioctl()* operations through the device files are supported for userspace applications. For instance, to allocate memory space on a specific PIM device (e.g., 40 KB memory allocation on the first PIM device), an application first opens the device file (e.g., `/dev/pim0`) and invokes the *ioctl()* system call with a predefined operation id (e.g., `PIM_IOC_GETPAGES`) and the amount of requesting space in the number of pages (e.g., 10). Similarly, to initiate a task execution on a PIM device, the application invokes *ioctl()* with a pointer to a structure containing target function and argument addresses, and a dedicated operation id (e.g., `PIM_IOC_EXECUTE`). In addition, the device driver supports *ioctl()* operations that allow host applications to access PIM-internal runtime information such as PIM core utilization and memory usage. In our design, the physical space allocation of PIM memory is managed by the device driver. Current allocation status of each PIM memory can be acquired by reading the */proc* entry.

C. Low-Level PIM Library

The low-level PIM library of AnalyzeThat is layered atop the PIM device driver and consists of a set of functions that allow users to manually control PIM devices. Its usage is similar to that of POSIX dynamic memory and pthread functions. The library primarily provides two functionalities—memory management and task offloading, by wrapping the low-level *ioctl* interface. For memory management, it provides *pimmalloc()* and *pimfree()*. Programmers can allocate memory using the *pimmalloc()* function similar to the standard *malloc()* function, with an additional argument of *pim_id* to specify a PIM device that the space is allocated from. Similar to the *malloc()* function that internally invokes the *brk()* system call to extend the data segment if needed, *pimmalloc()* requests the device driver via the aforementioned *ioctl()* interface to allocate memory pages and expand the data segment. Note that the device driver globally synchronizes memory allocation requests from multiple applications, and grants applications access to acquire a particular memory region. To offload tasks to the PIM cores, the library provides *pimexec_exec()*. Programmers can offload a function to a specific PIM core by providing the pointer of the function and a *pim_id*. For functions running in parallel on PIM cores, programmers can synchronize the tasks using the *pimexec_wait()*, which forces a wait for the specified thread to terminate.

Although the low-level library could have been implemented with an existing heterogeneous computing framework (i.e., OpenCL [23]), we adopt a PIM-specific framework since OpenCL is known to sacrifice the performance to provide portability among different hardware [24].

IV. PADS: AnalyzeThat Programming Interface

While the low-level library (§ III-C) provides direct access to the PIM devices to advanced programmers, the PIM-Aware Data Structure (PADS) is a higher-level data abstraction that hides the intricate details of the hardware. Thematic to PADS is the encapsulation of data, the analysis to be performed on the data and the mechanisms to overlay

both the data and the analysis on the PIM device array. Users only need to create and manipulate the data structure in order to take advantage of the PIM functionality, while remaining oblivious to the complexity of the hardware. To this end, the PADS abstraction is composed of two components, namely the *key-value container* and *parallel operations*.

A. PADS Key-Value Container

We have chosen to represent the data within PADS using a key-value container, i.e., data is stored as key-value pairs. Key-value pair representation of data is simple yet generic enough to be used for various data analysis applications. Various data analysis systems and modern databases adopt key-value pair representation [25], [26], [27]. Internally, a PADS key-value container consists of n sub-containers, each of which is associated with a single PIM device. To use the PADS data structure, an application first creates a PADS key-value container object (referred to as a PADS object, hereafter). Then, the application can simply put key-value pairs into the PADS object, similar to using other familiar data containers, e.g., C++ *queue*, *set*, *map*, etc. The application does not have to specify the sub-container that internally stores the key-value pair. Instead, the AnalyzeThat runtime transparently selects a sub-container based on a data placement policy, as we will explain further in § V-B.

B. PADS Operations

PADS supports a set of operations that run on the associated PADS object. This includes operations that can facilitate data analysis tasks on the PADS object. Many data analysis tasks in practice take input data from files. Manually parsing a file and converting raw data to a structured record set is a tedious, error-prone task. PADS provides an *import()* function that automatically parses a given input file and populates a PADS object with the parsed key-value pairs. It supports many popular formats such as txt, csv and netCDF. A user can specify his own parser function. Similarly, the *export()* function writes all key-value pairs in a PADS object to a file in a specified format. Moreover, PADS allows users to perform customized data processing via the popular *map()* and *reduce()* interface [25]. The *map()* function, a member function of the PADS object, takes a user-defined function and an output PADS object as arguments. The user-defined function is executed on every key-value pair in the PADS object. The *reduce()* function groups key-value pairs in a PADS object based on their keys and produces a single, reduced key-value pair for each key group.

V. AnalyzeThat runtime

When programming with the high-level PADS abstraction (§ IV), users can focus on writing application logic without having to understand the PIM hardware architecture. Below the PADS abstraction, the AnalyzeThat runtime transparently handles PIM hardware-specific details, i.e., data distribution and parallel task execution across the PIM array. Figure 2 shows the internal architecture of the runtime. Users can simply populate a PADS key-value container and perform parallel operations, e.g., *map()* and *reduce()*. The runtime consists of the Device Manager, Placement Manager and Task Manager, and internally handles all PADS operations with regard to the underlying PIM hardware architecture.

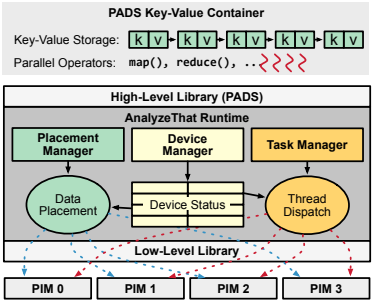


Figure 2: Users write applications with the PADS library, and the AnalyzeThat runtime transparently manages PIM-specific details.

A. Device Manager

To optimize the performance, the runtime should consistently keep track of the device status. For instance, the runtime should know which PIM core is busy before assigning tasks to the PIM device. Likewise, to evenly distribute data load, the runtime needs to track which PIM memory is most- or least-populated. To this end, the Device Manager in the runtime periodically gathers PIM device-specific information and stores each device's utilization statistics into a global device status table in memory. The records in the device status table are updated every 3 seconds, which is tunable. In particular, device-specific information, e.g., memory wearout status and PIM core utilization, is directly fetched from the device via the *ioctl()* system call (§ III-C). The device utilization statistics can be gathered directly from the PADS objects. The device status table is referenced by the Placement Manager and the Task Manager for making runtime decisions such as data placement and PIM thread dispatch, respectively.

B. Placement Manager

When an application inserts a key-value pair into a PADS container the Placement Manager selects a sub-container that is associated with a single PIM device. Particularly, the decision is made based on the following three objectives: 1) Balancing the load evenly across the PIM array. Load imbalance can slow down the overall execution time, which is determined by the completion time of the slowest PIM task. 2) Grouping key-value pairs in a single PIM device based on their keys. The *reduce()* operation runs faster if the key-value pairs are already grouped by their keys. In addition, the absence of proper aggregation may incur frequent remote PIM accesses, which can lead to a significant performance drop. 3) Storing output key-value pairs in a local PIM memory. Again, accessing local PIM memory is substantially faster than accessing remote PIM memory. Moreover, it eliminates potential lock contention among multiple PIM threads. Based on these objectives, the Placement Manager implements the following three *static* and one *dynamic* data placement policies. Round-Robin (RR) stores data in a round-robin order. Local-Assignment

(LA) always assigns a key-value pair to the same PIM device to maximize the local PIM memory accesses. Hashing (HS) effectively aggregates key-value pairs based on the key. Dynamic (DY) is a hybrid approach that takes advantage of the other three static policies.

C. Task Manager

The Task Manager is responsible for the execution of the PADS application program by utilizing the PIM-architecture, i.e., fast local memory access and parallelism. Specifically, to maximize the local PIM memory access, the Task Manager spawns a PIM thread on every PIM device. Each PIM thread then executes the user-defined PADS operation, which is tightly associated with the PADS key-value pairs in the local PIM memory. Internally, the runtime exploits the low-level PIM library functions (§ III-C) to accomplish task offloading. In addition, the Task Manager also optimizes the PADS operations based on workload characteristics.

VI. Evaluation

Implementation We have implemented an emulation framework for AnalyzeThat using 5,000 lines of C/C++ code. We emulated the PIM device by preallocating the system memory and binding threads to cores. To emulate the different access characteristics of the PIM memory, we introduced delays according to the memory access types, i.e., access from the host core or the PIM core. Specifically, a thread stays in a busy loop until the processor timestamp counter (TSC) reaches a desired value. The delay is computed based on a relative delay from a baseline when the PIM core accesses its local memory. When the PIM core accesses remote PIM memory, we add a delay corresponding to the time taken to access remote memory in a NUMA node. Our measurements of local and remote memory bandwidth on a NUMA node are 6,733.7 MB/s and 4,567.5 MB/s, respectively, i.e., remote memory is 32.2% slower than local memory. When the PIM accesses the remote memory, either the DRAM on the host or memory on another PIM, we add a 32% delay. Also, in our setup the DRAM access is 4× slower than PIM's local memory access [9].

Testbed Our test machine comprised of two processors (1.8 GHz Intel Xeon E5-2603, each with four cores), 64 GB RAM and ran the RedHat Enterprise Linux 6.5 with the 3.1.22 kernel. We dedicated one core for the host-side processing and emulated up to seven PIM devices with the remaining seven cores. For each emulated PIM device, we preallocated 4 GB of host memory and decreased the clock speed of the core to 1.2 GHz.

A. Programmability of PADS

First, we demonstrate the effectiveness of the PADS library. Using the PADS library, we implemented four representative big data applications in a few tens of lines of code. Here, we briefly explain each application and our implementation.

GroupByAggregation (GAG) computes the statistical summary, e.g., the total sum and average value from numerical datasets. Each line of the input file is composed of key-value pairs, delimited by a comma (","). The program first parses the input file and produces a set of key-value pairs. It then performs calculations by grouping the key-value pairs on the same key. Figure 3 shows the code snippet of our implementation. In the code snippet, *aggMap()* function appends "1" to the value of each key-value pair in *data*, and inserts the key-value pair into *mapped*. Then, *aggReduce()* performs calculations by aggregating all values in *mapped* based on their keys and stores the result in *result*.

```
// Group-by-Aggregation and Aggregation
void aggMap(char *k, char *v, PADS& t) {
    strcpy(new_val, v);
    strcat(new_val, "1");
    t.insert(k, new_val); // "A" instead of k for AG
}
char *aggReduce(char *k, char *v, char *reduced) {
    tokenizer(v, head, tail, "␣");
    tokenizer(reduced, sum, int, avg, "␣");
    sum = stoi(sum) + stoi(head)
    cnt = stoi(cnt) + stoi(int)
    avg = (double) sum / cnt;
    sprintf(reduced, "%d␣%d␣%f", sum, int, avg);
}
int main(void) {
    PADS data, mapped, result;
    data.import("input.txt");
    data.map(mapped, aggMap);
    mapped.reduce(result, aggReduce);
}
```

Figure 3: Implementation overview of data analysis applications using PADS library.

Aggregation (AG) works similar to GAG but calculates the global statistical summary, i.e., not based on keys. We implemented AG with a slight modification to the GAG program. In particular, we replace key in every key-value pair with the same value (e.g., 'A') in *aggMap()*, to make all key-value pairs share the same key. Therefore, *aggReduce()* calculates the statistical summary of all key-value pairs.

Grep (GR) is a string matching application that prints lines containing a matching keyword from the input file.

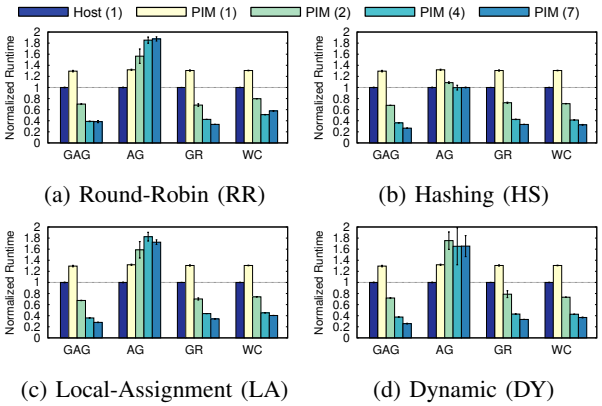
WordCount (WC) counts the occurrences of each word in the input text file. Each line is parsed as a $< line\_no, text >$ pair. *wcMap()* then generates $< word, 1 >$ for each pair, which is appended to *target*.

For the rest of this section, we use these four applications to study the performance of AnalyzeThat. In particular, for AG and GAG, we synthetically generated the input data consisting of 100 million entries (850 MB) with a normal distribution. For GR and WC, we used a 6.4 GB text file that concatenates contents of a wiki site [28]. Each experiment was repeated five times, and we report the average with the 95% confidence interval.

B. AnalyzeThat Performance

We compared the performance of AnalyzeThat against a host-based approach for running the applications from § VI-A For AnalyzeThat, we also evaluated four different data placement policies, i.e., Round-Robin (RR), Hashing (HS), Local-Assignment (LA) and Dynamic (DY) (§ IV).

Figure 4 shows the application runtimes of AnalyzeThat and the host-based approach. The results are normalized to the runtimes of the host-based approach. We observe that AnalyzeThat with a single PIM (PIM(1)) is 15-30%



(a) Round-Robin (RR) (b) Hashing (HS)

(c) Local-Assignment (LA) (d) Dynamic (DY)

Figure 4: Comparison of AnalyzeThat for different data placement policies, *without* Local Reduce. DY does not run wearout-aware algorithm ($w = 0$). Host(1) refers to the host with a single CPU, and PIM(n) refers to n PIM devices connected to a single host. Runtimes of Host(1) for GAG, AG, and GR were 405.7, 356.8, 189.8, and 44.1 seconds, respectively.

slower than the host-based approach with a single core (Host(1)). Even though the PIM internal memory bandwidth is greater than the host DRAM bandwidth, most workloads are compute-intensive, and thus the more powerful host CPU outweighs the higher memory bandwidth. However, AnalyzeThat with two PIM devices (PIM(2)) outperforms the host-based approach by 20-30%. In addition, AnalyzeThat scales almost linearly as we use more PIM devices, except for AG. For AG, the runtimes increase as more PIMs are used due to its workload characteristics (Figure 4(a), (c) and (d)). In addition, HS shows a different trend from the other policies for the AG workload. This is because HS places all intermediate data with the same key into a single PIM device so that it can avoid lock contentions during the reduce task.

For GR, we observe little performance variance across the different data placement policies. This is because GR is implemented only with *map()* operations (§ VI-A) and not affected by the performance variance of the *reduce()* operation, i.e., the lock contention and skewed distribution of the intermediate data.

We expect that further optimization can be achieved by adopting the Local Reduce (LR) technique, in which each PIM thread performs the *reduce()* operation locally by creating an intermediate PADS object in the local PIM memory. After Local Reduce, the global *reduce()* takes the sorted key-value pairs in the intermediate PADS objects to create the final result. We will perform further experiments on the effect of Local Reduce in future work.

VII. Concluding Remarks

We have developed AnalyzeThat as a means to abstract the complexity of PIM hardware, and to reduce the programming efforts needed to use the same. We have shown how representative data analysis applications can be effectively implemented using PADS, a high-level data and programming abstraction. PADS enables the use of an array of PIM devices as a key-value container, and applies a set of operations on the container based on an intelligent runtime system. Together, these concepts alleviate the effort required to program the PIM hardware.

Acknowledgments

This research was supported in part by the U.S. DOE's Office of Advanced Scientific Computing Research (ASCR) under the Scientific data management program, and the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MISP) (No. 2015R1C1A1A0152105). The work was also supported by, and used the resources of, the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at ORNL, which is managed by UT Battelle, LLC for the U.S. DOE, under the contract No. DE-AC05-00OR22725.

References

[1] P. M. Kogge, J. B. Brockman, T. Sterling, and G. Gao, "Processing in Memory: Chips to Petaflops," in *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA*, vol. 97, 1997.
[2] R. C. Murphy, P. M. Kogge, and A. Rodrigues, "The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems," in *Intelligent Memory Systems*, 2001.
[3] J. Adibi, T. Barrett, S. Bhatt, H. Chalupsky, J. Chame, and M. Hall, "Processing-In-Memory Technology for Knowledge Discovery Algorithms," in *Proceedings of the DaMoN*, 2006.
[4] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge, "A Low Cost, Multithreaded Processing-In-Memory System," in *Proceedings of the WMPI*, 2004.
[5] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *Proceedings of the ICCD*, 2012.
[6] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang *et al.*, "The Architecture of the DIVA Processing-In-Memory Chip," in *Proceedings of the SC*, 2002.
[7] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-Stacked Memory Logic Devices on MapReduce Workloads," in *Proceedings of the ISPASS*, 2014.
[8] M. Scrbak, M. Islam, K. M. Kavi, M. Ignatowski, and N. Jayasena, "Processing-in-Memory: Exploring the Design Space."
[9] 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 HPDC*, 2014.
[10] "Micron's Automata," https://www.micronautomata.com.
[11] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung *et al.*, "Phase-Change Random Access Memory: A Scalable Technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, 2008.
[12] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The Missing Memristor Found," *nature*, vol. 453, no. 7191, 2008.
[13] A. Driskill-Smith, "Latest Advances and Future Prospects of STT-RAM," in *Proceedings of the NVMW*, 2010.
[14] M. Islam, M. Scrbak, K. M. Kavi, M. Ignatowski, and N. Jayasena, "Improving Node-Level MapReduce Performance Using Processing-in-Memory Technologies," in *Proceedings of the Euro-Par*, 2014.
[15] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, M. Meswani, M. Nutter, and M. Ignatowski, "A New Perspective on Processing-in-memory Architecture Design," in *Proceedings of the SIGPLAN*, ser. MSPC '13, 2013, pp. 7:1–7:3.
[16] J. Dongarra *et al.*, "The International Exascale Software Project Roadmap," *International Journal of High Performance Computing Applications*, 2011.
[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "SkewTune: Mitigating Skew in MapReduce Applications," in *Proceedings of the SIGMOD*, 2012.
[18] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System," in *Proceedings of the IISWC*, 2009.
[19] "Cache Coherent Interconnect for Accelerators (CCIX)," http://www.ccixconsortium.com.
[20] S. Nobis, "AMDs Unified CPU & GPU Processor Concept."
[21] G. Loh, N. Jayasena, M. Oskin *et al.*, "A Processing in Memory Taxonomy and a Case for Studying Fixed-function PIM," in *Near-Data Processing Workshop*, 2013.
[22] "Netezza Data Warehouse — IBM," https://www.ndm.net/datawarehouse/IBM/netezza.
[23] "OpenCL - The open standard for parallel programming of heterogeneous systems," https://www.khronos.org/opencl/.
[24] J. Fang, A. L. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *2011 International Conference on Parallel Processing*, 2011.
[25] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.
[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *Proceedings of the ACM SIGOPS*, vol. 41, no. 6, 2007.
[27] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High Throughput Persistent Key-Value Store," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, 2010.
[28] "EnWiki.NET: Encyclopaedia Britannica Ultimate," http://www.enwiki.net/.