Performance Implications of Processing-in-Memory Designs on Data-Intensive Applications

Borui Wang⋆ bwang27@ucsc.edu
Martin Torres† mtorres58@ucmerced.edu
Dong Li† dli35@ucmerced.edu
Jishen Zhao⋆ jishen.zhao@ucsc.edu
Florin Rusu† frusu@ucmerced.edu

†University of California, Merced ⋆University of California, Santa Cruz

***Abstract*—The popularity of data-intensive applications and recent hardware developments drive the re-emergence of processing-in-memory (PIM) after earlier explorations several decades ago. To introduce PIM into a system, we must answer a fundamental question: what computation logic should be included into PIM? In terms of computation complexity, PIM can be either relatively simple, fixed-functional, or fully programmable. The choice of fixed-functional PIM and programmable PIM has direct impact on performance. In this paper, we explore the performance implications of fixed-functional PIM and programmable PIM on three data-intensive benchmarks—including a real data-intensive application. Our results show that – with PIMs – we obtain 2.09x-91.4x speedup over no PIM cases. However, the fixed-functional PIM and programmable PIM perform differently across applications (with performance difference up to 90%). Our results show that neither fixed-functional PIM nor programmable PIM can perform optimally in all cases. We must decide the usage of PIM based on the characteristics of the workload and PIM (e.g., instruction-level parallelism), and the PIM overhead (e.g., PIM initialization and synchronization overhead).**

I. Introduction

Data-intensive applications, including scientific simulations and enterprise on-line services, generate massive volumes of data and demand superior data processing capabilities in terms of speed and automation. The demands from in-situ data processing, such as scientific visualization for real-time analysis and computational finance for stock analysis, further impose heavy requests on the efficiency of data movement and processing.

To address the above problems, processing-in-memory (PIM) re-emerges after early explorations a few decades ago [10], [13], [29], [23], [16], [8], [22], [26], [28]. By adding the data processing capability into memory, PIM can reduce the latency and energy consumption associated with data movement through the cache and memory hierarchy by moving computation closer to data. Furthermore, PIM enables data processing in parallel with the CPU, which introduces new parallelism into the system.

The re-emergence of PIM is strongly driven by recent hardware developments and applications. At the hardware level, the advancement of mixed logic, memory processes, and die stacking make the implementation of PIM easier and cheaper. At the application level, data-intensive applications with irregular access patterns (e.g., graph traversals and certain multi-phase, multi-stage scientific simulations) make data movement more frequent and expensive. PIM is well positioned to leverage new hardware techniques to address the emerging application problems.

To introduce PIM into the system, we must answer a fundamental question: what computation logic should be included into PIM? In terms of computation complexity, PIM can be either relatively simple, fixed-functional, or fully programmable. Fixed-functional PIMs can implement specific operations. For example, some PIMs shift a long user-specified contiguous region along a short, fixed distance in memory [20]; some PIMs implement complex regular expressions [7]. Fully programmable PIMs have fully programmable logic in memory and have the expressiveness and flexibility of a conventional processor (or configurable logic device). Such PIMs include DIVA [17] and FlexRam [12].

The choice of fixed-functional PIM and programmable PIM has direct impact on performance, power/energy, hardware area size, and the interface with software. In particular, the fixed-functional PIM has relatively simple logic, less power consumption and area size than the programmable PIM, but it has constraints on the computation capabilities and can suffer from performance loss due to (1) CPU-PIM or PIM-PIM synchronization; (2) data movement; (3) task spawning overhead. Some of the performance loss can be avoided with programmable PIMs. In contrast, programmable PIMs can offload coarse-grained tasks from CPU, overlapping PIM computation with CPU tasks and reducing the performance overhead suffered in the fixed-functional PIM. However, programmable PIMs have large area size which makes it difficult to achieve

massive parallelism as in the fixed functional PIM; they also have larger power consumption.

In this paper, we explore the performance implications of the fixed-functional PIM and programmable PIM. We use two benchmarks and one real data-intensive application. One benchmark performs intensive sparse linear algebra, and the other performs specific stencil computation. These two benchmarks are representative and can be one of the major workloads offloaded to PIM in the future. The data-intensive application comes from the data analytics domain and includes gradient computation over massive datasets. We rely on a PIN-based emulation infrastructure to study the implication of the fixed-functional PIM and programmable PIM on these benchmarks and the application. Our results show that with PIMs, we have 2.09x-91.4x speedup over no PIM cases. However, the fixed-functional PIM and programmable PIM perform differently across applications (with performance difference up to 90%). Our results show that neither fixed-functional PIM nor programmable PIM can perform optimally in all cases. We must decide the usage of PIM based on the characteristics of workload and PIM (e.g., instruction-level parallelism) and the PIM overhead (e.g., PIM initialization and synchronization overhead).

II. Methodology

We investigate the performance of various PIM designs by studying the instruction traces of a set of workloads that are likely to benefit from offloading data-intensive operations to the memory side. We use PIN [25], a dynamic instrumentation-based program analysis tool to collect the traces of application execution. To generate the traces, we run our applications on a machine configured with 8 Intel Core i7-4790 cores running at 3.6 GHz. The machine has two 8 GB DDR3-1600 DRAMs. Table I summarizes the key architectural parameters of the system.

Table I: Architectural parameters of the machine where we catch the application traces.

Processor	32nm Intel Core i7 operating at 3.6GHz
Cores	8 OoO cores operating at 3.6GHz
L1 Cache (Private)	64KB, 4-way, 64B lines
L2 Cache (Private)	256KB, 8-way, 64B lines
L3 Cache (Shared)	Multi-banked, 8MB, 16-way, 64B lines
Memory	DDR3-1600, 8GB, 8 banks,

Our simulation framework models the performance of a system with fixed-functional PIMs and a programmable PIM. Figure 1 illustrates the components of our simulation framework. We investigate the case where the memory system can integrate many fixed-functional PIMs (50 adders and 50 multipilers in total), whereas it can only adopt a single programmable PIM. This is because the programmable PIM typically requires complex functional components and logic; with the limited area and power budget of the memory, we cannot integrate multiple programmable PIMs.

Our trace-based simulation framework is sufficiently flexible to allow us to explore various performance parameters related to PIM designs. In particular, Table II lists the performance parameters that we employ in our simulation. We assume that the memory clock frequency is half of the CPU clock frequency. Therefore, operations that typically take one CPU cycle will take two cycles on the memory side. The programmable PIM typically performs more complex operations with larger input and output data structures than the fixed-functional PIM. Therefore, we adopt a longer latency for initializing and returning from programmable PIM operations. We also introduce PIM synchronization latency, which is 200 CPU cycles. The PIM synchronization latency includes: 1) CPU acquisition of a PIM lock (one memory access, i.e., 100 CPU cycles); 2) PIM releasing the lock (0 CPU cycle); 3) CPU examining whether the lock is released (one memory access, i.e., 100 cycles).

To specify the code regions offloaded to the PIMs, we introduce a set of annotations shown in Figure 2. For any annotated code regions, our PIN-based emulator will collect traces and model their performance on PIMs based on the performance parameters.

III. Applications

We use two benchmarks (CG and MG) and one real application (GLADE) to evaluate the performance of the fixed-functional PIM and programmable PIM.

A. Conjugate Gradient (CG)

This benchmark comes from the NAS parallel benchmark 3.3 [4]. It uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non-zeros. Figure 3 shows the pseudocode of the major computation loop in this benchmark.

The computation of CG is dominated by a multiplication-addition operation represented as $a = b + c * d$ (see lines 10, 16, 24, 25, 30, 36, and 43). In many cases, a, b, c and/or d in the operation are the elements of specific vectors or matrices. Furthermore, the access to the vector p (line 10) and the vector z (line 43) come from indirect data references (e.g., $p[colidx[k]]$ and $z[coldix[k]]$). These accesses can be random and have poor data locality. The memory access pattern of CG with indirect data references is because of the compressed row storage (CRS) format for storing sparse matrices. The index array $colidx$ indicates the positions of non-zero elements in the sparse matrices. Using the

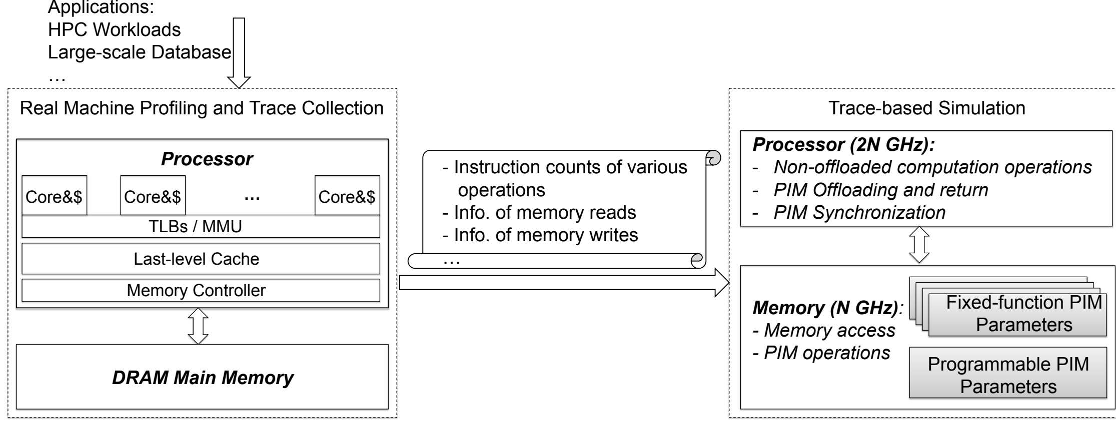

Figure 1: Overview of our simulation framework.

Table II: PIM parameters employed in our trace-based simulation, assuming memory clock frequency is half of the CPU clock frequency.

Parameter	Description	Value
$t_{CPUcomp}$	CPU computation instruction latency	Profiled on real machine
t_{mem}	Latency of executing a memory load/store instruction on CPU	100 CPU cycles
$t_{fixPIMcomp}$	Fixed-function PIM computation instruction latency	$2\times$ $t_{CPUcomp}$ of the same instruction
$t_{progPIMcomp}$	Programmable PIM computation instruction latency	$2\times$ $t_{CPUcomp}$ of the same instruction
$t_{fixPIMoffload}$	Latency of initializing the operations to be offloaded to a fixed-function PIM	2 CPU cycles
$t_{fixPIMreturn}$	Latency of returning fixed-function PIM operation results to CPU	2 CPU cycles
$t_{progPIMoffload}$	Latency of initializing the operations to be offloaded to the programmable PIM	10 CPU cycles
$t_{progPIMreturn}$	Latency of returning programmable PIM operation results to CPU	10 CPU cycles
t_{sync}	Latency of synchronizing multiple PIMs	200 CPU cycles

```
fixPIM_begin();
Operations that are offloaded to fixed-function PIMs
fixPIM_end();
```

```
progPIM_begin();
Operations that are offloaded to the programmable PIM
progPIM_end();
```

Figure 2: The software interface for identifying the group of operations to be offloaded to PIMs.

array $colidx$ to reference data elements in the vectors can avoid unnecessary computation. The memory access pattern with indirect data references is common in sparse linear algebra. Because of the poor data locality in this memory access pattern, the traditional CPU-based computation can cause lots of cache misses and frequent data movement between CPU and main memory.

We offload the primitive multiplication-addition operation into the fixed-functional PIMs. We assume that the fixed-functional PIMs can use the index array $colidx$ to access data elements for the computation. For the programmable PIM, we offload the whole computation loop shown in Figure 3.

B. Multi-Grid (MG)

This benchmark comes from the NAS parallel benchmark 3.3 [4]. It approximates the solution to a three-dimensional discrete Poisson equation using the V-cycle multi-grid method on a rectangular domain with periodic boundary conditions. In the V cycle, the computation starts from the finest refinement level, going down level by level toward the bottom, then back up to the top. The V-cycle multi-grid method involves applying a set of stencil operations sequentially on the grids at each level of refinement [35]. The stencil operations happen in various execution phases, including restriction, prolongation, evaluation of residual, and point relaxation. Figure 4 shows the pseudocode of some stencil operations in the evaluation of residual.

The stencil operations in MG often involve a 4-point stencil. Figure 4 shows such typical stencil operations. For example, to calculate u1[i1], we must access four neighbor points (u[i1][i2-1][i3], u[i1][i2+1][i3], u[i1][i2][i3-1] and u[i1][i2][i3+1]) which form a sten-

```
1:  z = 0
2:  r = x
3:  ρ = r^T r
4:  p = r
5:  for i=0,1,... do
6:      //computing q = Ap
7:      for j=1,lastrow-firstrow+1 do
8:          sum = 0.0
9:          for k=rowstr[j],rowstr[j+1]-1 do
10:             sum = sum + a[k] * p[colidx[k]]
11:         end forq[j] = sum
12:     end for
13:
14:     //computing α = ρ/(p^T q)
15:     for j=1,lastcol-firstcol+1 do
16:         d = d + p[j] * q[j]
17:     end for
18:     α = ρ/d
19:     ρ0 = ρ
20:
21:     //computing z = z + αp and r = r − αq
22:     ρ = 0.0
23:     for j=1,lastcol-firstcol+1 do
24:         z[j] = z[j] + α * p[j]
25:         r[j] = r[j] − α * q[i]
26:     end for
27:
28:     //computing ρ = r^T r
29:     for j=1,lastcol-firstcol+1 do
30:         ρ = ρ + r[j] * r[j]
31:     end for
32:     β = ρ/ρ0
33:
34:     //computing p = r + βp
35:     for j=1,lastcol-firstcol+1 do
36:         p[j] = r[j] + β * p[j]
37:     end for
38:
39:     //computing residual norm: ||r|| = ||x − A.z||
40:     for j=1,lastrow-firstrow+1 do
41:         d = 0.0
42:         for k=rowstr[j],rowstr[j+1]-1 do
43:             d = d + a[k] * z[coldix[k]]
44:         end forr[j] = d
45:     end for
46:     ...
47: end for
```

Figure 3: Pseudocode of the major computation loop in the CG benchmark. The blue lines indicate the computation offloaded to the fixed-functional PIM.

cil. We offload these stencil operations into the fixed-functional PIMs. For the programmable PIM, we offload the major computation routines (particularly mg3P and resid, not shown in the code excerpt due to the space limitation).

C. GLADE

GLADE [5] is a parallel data processing system for large scale data analytics. It executes tasks specified

```
1: for i3=2,n3-1 do
2:     for i2=2,n2-1 do
3:         for i1=1,n1 do
4:             u1[i1] = u[i1][i2-1][i3] + u[i1][i2+1][i3] +
               u[i1][i2][i3-1] + u[i1][i2][i3+1]
5:             u2[i1] = u[i1][i2-1][i3-1] + u[i1][i2+1][i3-1] +
               u[i1][i2-1][i3+1] + u[i1][i2+1][i3+1]
6:         end for
7:     end for
8: end for
```

Figure 4: A code excerpt from the residual computation ($r = v - AU$) in MG. The blue lines indicate the computation offloaded to the fixed-functional PIM.

using the abstract User-Defined Aggregate (UDA) interface [30]. In this application, we train a support vector machine (SVM) model using gradient descent optimization. We use the batch version of gradient descent (BGD) [31].

The pseudo-code for BGD when applied to SVM training is shown in Figure 5. The BGD algorithm takes as input the data examples and their labels, the loss function Λ, the gradient of the objective $\nabla\Lambda$, and the initial values for the model and step size. The optimal model is returned as output [32].

The main stages of this algorithm are the updates of the gradient computation, model, and step size. These are all executed until convergence. We are able to offload the vector multiplication seen on lines 12 and 18. Line 12 shows the dot product between the model, which is represented by a vector, and the current example. Line 18 shows the gradient update, which needs access to the gradient vector, model vector, and current example. Essentially, we iterate through the examples and update the gradient, which is minimizing our objective function in order to provide us with a better model. Because these methods are invoked many times while iterating through the examples, this vector multiplication becomes computationally expensive.

We offload the gradient computation in both the fixed-functional and programmable PIMs. Much like in the case with CG, we benefit from PIM because of the data access pattern and frequent data movement.

IV. Evaluation

We report the performance of the two NAS benchmarks and GLADE in this section.

A. NAS Benchmarks

Figure 6 shows the performance results for the two NAS parallel benchmarks, CG and MG. We notice that PIMs bring significant performance improvement: for CG there are 18.6x (fixed-functional PIMs) and 13.9x (programmable PIM) performance improvement over the non-PIM case; for MG there are 8.7x (fixed-functional

```
1:  // x[i] is example i
2:  // index[i] is the indexes for example i
3:  // y[i] is the label for example i
4:  // w is the model
5:  // g is the gradient
6:
7:  // iterate over all the training examples
8:  for (i = 1, numExamples) do
9:     // compute the dot-product between example i and
       model w
10:    s = 0
11:    for j = 0, size(x[i]) do
12:       s += x[i][j] * w[index[i][j]]
13:    end for
14:
15:    // finalize the gradient computation
16:    if (1 - y[i] * s > 0) then
17:       for j = 0, size(x[i]) do
18:          g[index[i][j]] += x[i][j] * y[i]
19:       end for
20:    end if
21: end for
```

Figure 5: SVM gradient computation. The blue lines show the operations offloaded to both fixed-functional and fully-programmable PIM

PIMs) and 91.4x (programmable PIM) performance improvement over the non-PIM case.

For CG, the workloads offloaded to the fixed-functional PIMs and programmable PIM are almost the same. However, the fixed-functional PIMs finish the offloaded workload in much shorter time than the programmable PIM (see "offloading computation execution time" in Figure 6). This is because the fixed-functional PIMs have larger number of operators (adders and multipliers) than the programmable PIM, potentially providing more comptuation parallelism than the programmable PIM. In general, for CG, the fixed-function PIMs perform 90% better than the programmable PIM.

For MG, we have a different story. The workload offloaded to the programmable PIM is larger than that offloaded to the fixed-functional PIMs. The specialized operations in the fixed-functional PIMs limits the offloadability of workloads. On the other hand, the programmable PIM has freedom to offload any workload, providing more opportunity to accelerate performance. Furthermore, the fixed-functional PIMs suffer from high PIM synchronization overhead (17% of the total execution time), while the programmable PIM has an ignorable synchronization overhead. In general, for MG, the programmable PIM performs 25% better than the fixed-functional PIMs.

In conclusion, neither fixed-functional PIMs nor programmable PIM can perform best in all cases. We must combine both of them to leverage their strength while avoiding their limitation.

B. GLADE

Figure 7 shows the results for GLADE. We first notice the big performance improvement brought by PIMs: we have 2.13x and 2.09x speedup with the fixed-functional PIMs and programmable PIM respectively. This is another demonstration of how PIMs can be beneficial for data-intensive applications. We further notice that the fixed-functional PIMs have better performance than the programmable PIM (2% performance improvement). Our further investigation reveals that the workloads offloaded to the fixed-functional PIMs and the programmable PIM are similar. The fixed-functional PIMs perform better because of massive numbers of operators supported in the PIMs. These operators provide larger computation parallelism than the programmable PIM. The programmable PIM cannot show better performance as MG does, because there is no instruction that is offloadable to the programmable PIM while not offloadable to the fixed-functional PIMs, hence the benefits of the programmable PIM can not be fully explored.

In general, the conclusion we get from the results of GLADE is consistent with that from MG and CG. There is no single PIM design that can stand out in all cases.

V. Related Work

A growing body of work has explored PIM design from various perspectives, such as architecture design [24], [14], [2], [3], [27], [9], [21], software interface [2], [1], OS and runtime [14], [36], and applications [15], [3]. These works can be grouped into two major categories – fixed-functional PIM [24], [14], [15], [2], [3], [27] and fully-programmable PIM [9], [36], [1], [21] – based on how the PIM computing interfaces with the software. Previous work mostly falls in one of the two categories. Yet, our work shows that neither fixed-functional PIM nor programmable PIM fit all use cases. Even with a single application, different stages can favor different types of PIMs. In this section, we present recent research in the two categories.

A. Fixed-Functional PIM Design and Management

This class of PIMs provides pre-defined or fixed functions with memory-side computing. Hardware and software can dynamically allocate or statically map instructions to the memory-side computing units. However, they are by definition non-programmable. Because functions are fixed, they are typically simple and general enough such that they can be reused heavily throughout the execution of various applications. In fact, a recent industry proposal – the HMC 2.0 standard [18] – supports a set of fixed-functional operations, including addition, bitwise swap, bit write, boolean operations, and numerical comparison. Yet, to keep the functions simple, limitations exist. For example, to minimize the effort

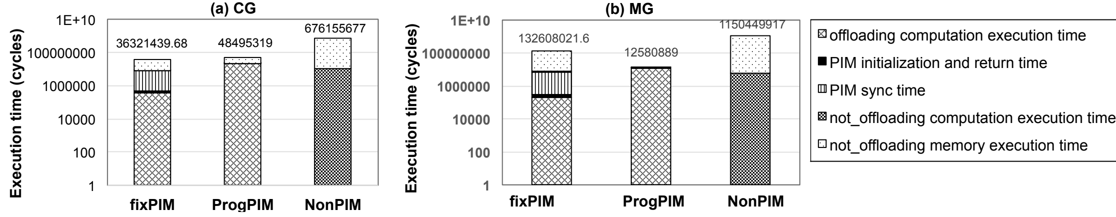

Figure 6: Preliminary results for two NPB benchmarks with emulation of fixed-function and programmable PIM.

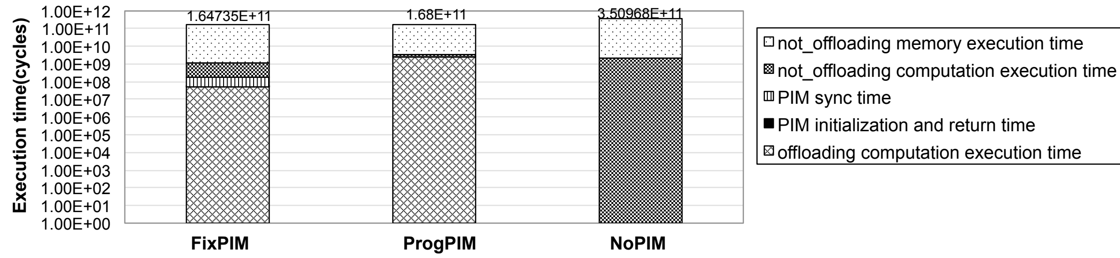

Figure 7: Preliminary results for GLADE with emulation of fixed-function and programmable PIM.

of converting original program segments into HMC operations, the target applications have to contain a significant amount of irregular memory accesses triggered by short code sections that can be easily spotted. As such, it especially benefits a subset of workloads, e.g., graph traversal applications, which issue a large number of irregular memory accesses mostly coming from a few lines of code. Programming of fixed-functional PIMs typically relies on low-level APIs [19] and data structures [6]. For instance, Chu et al. [6] introduce a PIM-aware data structure that uses low-level PIM APIs to allocate memory and keeps track of data and task mapping.

B. Programmable PIM Design and Management

Prior PIM work has focused on integrating programmable logic in memory modules [28], [8]. These PIMs can be programmed by application or system software in a similar way as conventional CPUs. As such, this design offers the expressiveness and flexibility of a CPU or configurable logic device, while reducing the overhead of off-chip data movement. Recent studies revisit the programmable PIM design in the context of the new HMC technology and new requirements of data-intensive applications. Most of these studies strive to integrate GPU-like computing units in memory [36]. Several studies employed ARM cores as computing units [1]. These programmable PIMs are well-suited for intrinsically parallel tasks like traversing large data structures by supporting many threads in flight simultaneously.

The programming model of the programmable PIM evolves from parallel programming models, such as OpenMP. For example, Fraguela et al. [11] and Solihin et al. [33] propose a series of directives and clauses that enrich the semantics of OpenMP directives. These directives provide great flexibility for task partitioning. Hall et al. [17] introduce a primitive host-to-memory interface that uses an event based mechanism to communicate computation to memory. Tseng et al. [34] introduce a data-triggered multithreading programming model that enables flexible expression of data-computation associations. As a result, their runtime triggers PIM computation upon update to the associated data. Ahn et al. [1] triggers computation in PIM as a (non-)blocking remote function call via message passing.

VI. Conclusions

This paper is a preliminary study on the performance implications of fixed-functional and programming PIM on data intensive applications. The paper reviews the limitations and strengths of the two PIMS from multiple perspectives. The fixed-functional PIM has more computation operators (e.g., adders and multipliers) than the prorammable PIM, hence can potentially provide more computation parallelism for specific operations; the programmable PIM provides greater flexibility to offload workloads than the fixed-functional PIM hence can provide better performance when there are no sufficient operations to offload to the fixed-functional PIM. The fixed-functional PIM also suffers from the synchronization overhead more frequently, because it provides workload offloading at finer granularity than the programmable PIM. We reveal that neither the fixed-functional PIM nor programmable PIM can perform optimally, and we must combine the two PIMs to achieve

best performance in all cases.

References

[1] 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*, pages 105–117, 2015.

[2] 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*, pages 336–348, 2015.

[3] B. Akin, F. Franchetti, and J. C. Hoe. Data reorganization in memory using 3d-stacked dram. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pages 131–143, 2015.

[4] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report 103863, NASA, July 1993.

[5] Y. Cheng, C. Qin, and F. Rusu. GLADE: Big Data Analytics Made Easy. In *SIGMOD 2012*.

[6] M. L. Chu, N. Jayasena, D. P. Zhang, and M. Ignatowski. High-level programming model abstractions for processing in memory. *Workshop on Near-Data Processing*, 2013.

[7] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. Architecture for Parallel Automata Processing. *IEEE Transaction on Parallel and Distributed Systems*, (12), 2014.

[8] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca. The architecture of the DIVA processing-in-memory chip. In *Proceedings of the 16th International Conference on Supercomputing*, pages 14–25, 2002.

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

[10] D. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie. Computational RAM: Implementing processors in memory. *IEEE Des. Test*, 16(1):32–41, Jan. 1999.

[11] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas. Programming the flexram parallel intelligent memory system. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, 1999.

[12] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas. Programming the flexram parallel intelligent memory system. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[13] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: The terasys massively parallel PIM array. *Computer*, 28(4):23–31, Apr. 1995.

[14] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti. 3D-stacked memory-side acceleration: Accelerator and system design. In *WoNDP*, pages 1–6, 20se14.

[15] Z. Guz, M. Awasthi, V. Balakrishnan, M. Ghosh, A. Shayesteh, and T. Suri. Real-time analytics as the killer application for processing-in-memory. In *WoNDP*, pages 1–3, 2014.

[16] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

[17] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping irregular applications to diva, a pim-based data-intensive architecture. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 1999.

[18] HMCC. Hybrid memory cube specification 2.0. http://http://www.hybridmemorycube.org/.

[19] T. Kelly, H. Kuno, M. Pickett, H. Boehm, A. Davis, W. Golab, G. Graefe, S. Harizopoulos, P. Joisha, A. Karp, N. Muralimanohar, F. Perner, G. Medeiros-Ribeiro, G. Seroussi, A. Simitsis, R. Tarjan, and S. Williams. Sidestep: Co-designed shiftable memory and software. Technical report, HP Labs, 2012.

[20] T. Kelly, H. Kuno, M. D. Pickett, H. Boehm, A. Davis, W. Golab, G. Graefe, S. Harizopulos, P. Joisha, A. Karp, N. Muralimanohar, F. Perner, G. Seroussi, A. Simitsis, R. Tarjan, and R. S. Williams. Sidestep: Co-designed shiftable memory and software. In *HP Technical Report*, 2012.

[21] C. D. Kersey, S. Yalamanchili, and H. Kim. Simt-based logic layers for stacked dram architectures: A prototype. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 29–30, 2015.

[22] Y. Kim, T.-D. Han, S.-D. Kim, and S.-B. Yang. An effective memory–processor integrated architecture for computer vision. In *Proceedings of the International Conference on Parallel Processing*, pages 266–, 1997.

[23] P. M. Kogge. EXECUBE-a new architecture for scaleable MPPs. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01*, pages 77–84, 1994.

[24] 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*, pages 1–6, 2013.

[25] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, Chicago, Illinois, June 2005.

[26] R. C. Murphy, P. M. Kogge, and A. Rodrigues. The characterization of data intensive memory workloads on distributed PIM systems. In *Revised Papers from the Second International Workshop on Intelligent Memory Systems*, pages 85–103, 2001.

[27] 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*, pages 258–261, 2015.

[28] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192–203, 1998.

[29] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, Mar. 1997.

[30] F. Rusu and A. Dobra. GLADE: A Scalable Framework for Efficient Analytics. *OS Review*, 46(1), 2012.

[31] F. Rusu and C. Qin. Speculative Approximations for Terascale Analytics. *CoRR 1501.00255*, 2014.

[32] F. Rusu, C. Qin, and M. Torres. Scalable Analytics Model Calibration with Online Aggregation. *IEEE Data Engineering Bulletin*, 38(3), 2015.

[33] Y. Solihin, J. Lee, and J. Torrellas. Automatic code mapping on an intelligent memory architecture. *IEEE Transactions on Computers*, 2001.

[34] H.-W. Tseng and D. M. Tullsen. Data-triggered multithreading for near data processing. In *Workshop on Near-Data Processing*, 2013.

[35] T. Wen. Introduction to the x10 implementation of npb mg. In *IBM Technical Report*, 2006.

[36] 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*, pages 85–98, 2014.