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

Borui Wang* **Martin Torres**

Dong Li **Jishen Zhao*** **Florin Rusu**

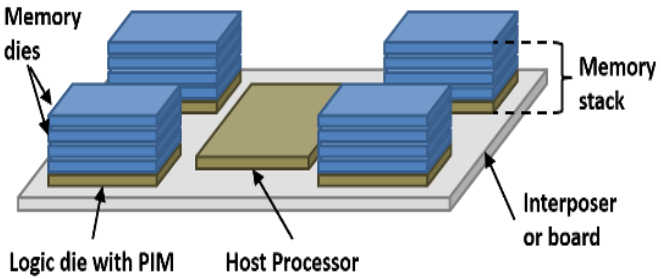
University of California, Merced * University of California, Santa Cruz

Re-emergence of Processing-in-Memory (PIM)

- Processing-in-memory addes data processing capability into memory
- The re-emergence of PIM is driven by hardware developments and applications
  - Hardware: the advancement of mixed logic, memory processes, and die stacking makes implementation easier and cheaper
  - Application: data intensive applications with irregular access patterns
  - The benefits of PIM: reduce latency and energy consumption; introduce new parallelism into systems

Two types of PIM

- Fixed-functional PIM
  - Implement a specific operations in memory
  - For example, shifting a long contiguous region along a short, fixed distance in memory
  - For example, complex regular expression



- Fully programmable PIM
  - Fully programmable logic in memory
  - have the expressiveness and flexibility of a conventional processor or configurable logic devices
  - For example, DIVA (USC & ND)
  - For example, FlexRam (UIUC)

Trade-offs of the two types of PIM

- The choice of fixed-functional PIM and programmable PIM has direct impact on performance, power/energy, hardware area size, and the interface with software

- Fixed-functional PIM
  - Pros: Simple logic, less power consumption and area size
  - Cons: constraints on the computation capabilities and performance overhead
    - Frequent sync
    - Frequent data movement
    - Task spawning overhead

Trade-offs of the two types of PIM

- Fixed-functional PIM
  - Pros: Simple logic, less power consumption and area size
  - Cons: constraints on the computation capabilities and performance loss
    - Frequent sync
    - Frequent data movement
    - Task spawning overhead

- Fully-programmable PIM
  - Pros: offloading coarse-grained tasks from CPU;
    - Overlapping PIM computation with CPU
    - Reducing the performance overhead in the fixed-functional PIM
  - Cons: larger area size and power consumption
    - Less parallelism

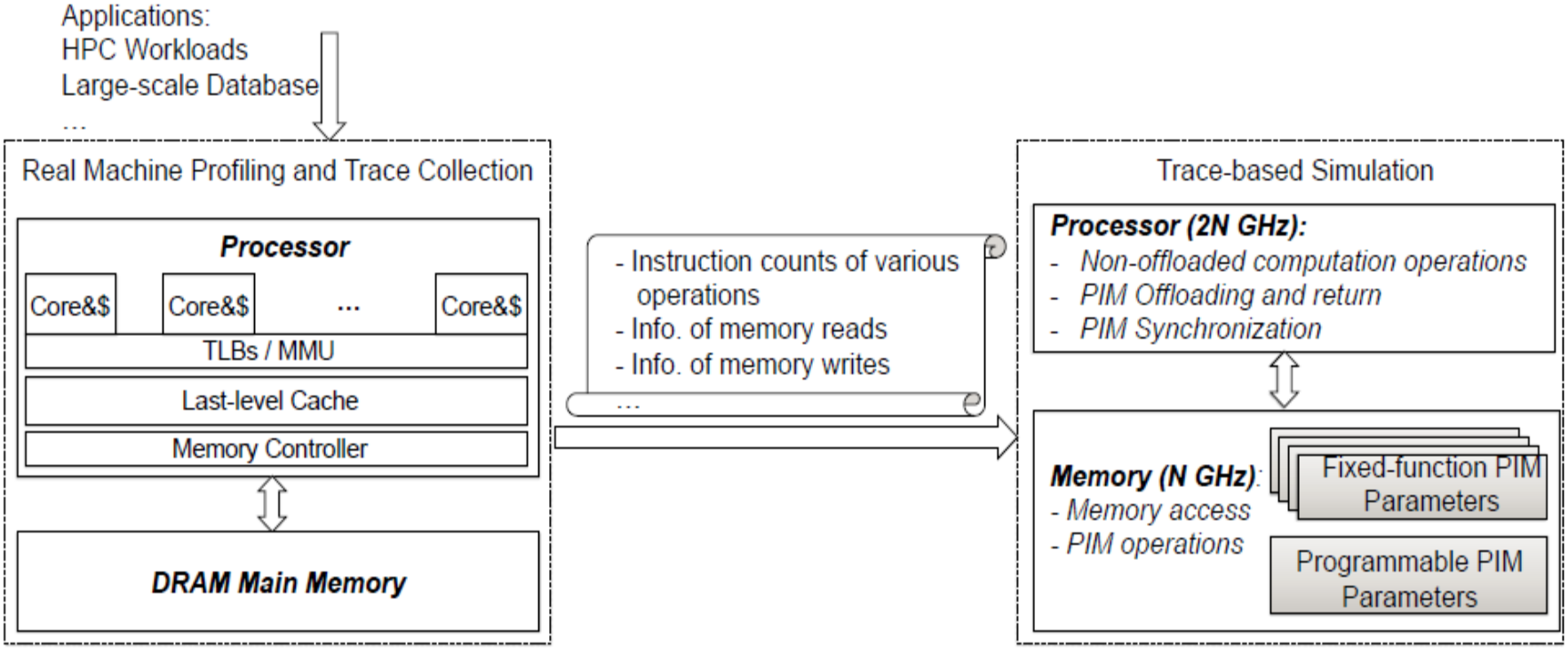
Story in a nutshell

- We study the performance implications of the fixed-functional PIM and programmable PIM on applications
  - We focus on performance in this paper
  - Thermal analysis can be found in another paper
    - Yuxiong Zhu, Borui Wang, Dong Li, and Jishen Zhao. Integrated Thermal Analysis for Processing In Die-Stacking Memory. In International Symposium on Memory Systems, 2016

- Evaluate two benchmarks and one real data-intensive applications
  - CG
  - MG
  - GLADE

Evaluation Methodology

- Collect the instruction traces of the workloads and use a trace-based simulation
  - Sufficiently flexible to allow us to explore various performance parameters
- Many fixed-functional PIMs vs. a single programmable PIM
  - 50 adders and 50 multipliers for fixed functional PIM



Simulation Parameters

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

- The memory clock frequency is half of the CPU clock frequency
- The programmable PIM has a longer latency for initializing and returning from programmable PIM operations
- PIM sync latency: 200 CPU cycles
  - CPU acquisition of a PIM lock (one memory access)
  - PIM releasing the lock
  - CPU examine whether the lock is released (one memory access)

Software interface

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

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

- We use a set of annotations to specify the code regions offloaded to PIMs
- We develop a PIN-based emulator to collect traces and model their performance on PIMs based on the performance parameters

Benchmark: NPB CG

- Conjugate gradient (CG)
  - 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
- The computation of CG is dominated by a multiplication-addition operation represented as $a = b + c * d$
- The access to the vectors p and z are based on indirect data references
  - Can be random and have poor data locality
  - The traditional CPU-based computation can cause lots of cache misses and frequent data movement between CPU and main memory

Benchmark: NPB CG

- Offload the primitive multi-addition operations into fixed functional PIMs
- For the programmable PIM, we offload the whole computation loop

```
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
```

Benchmark: NPB MG

- Multi-grid
  - Approximates the solution to a three-dimensional discrete Poisson equation using the V-cycle multi-grid method

- Workload characteristics
  - Apply a set of stencil operations sequentially on the grids
  - The stencil operations happen in various execution phases, such as restriction, prolongation, evaluation of residual, and point relaxation

Benchmark: MG

```
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.

- The stencil operations often involve a 4-point stencil
- We offload those stencil operations to the fixed functional PIMs
- For the programmable PIM, we offload the major computation routines (*mg3P* and *resid*)

Application: GLADE

- GLADE is a parallel data processing system for large-scale data analytics
  - In this application, we train a support vector machine (SVM) model using gradient descent optimization

- Workload characteristics
  - Takes the data examples as input
  - The main stage is the update of the gradient computation
  - Iterates through the examples and update the gradient

Application: GLADE

```
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

- We offload the gradient computation in both the fixed functional and programmable PIMs

Results: CG and MG

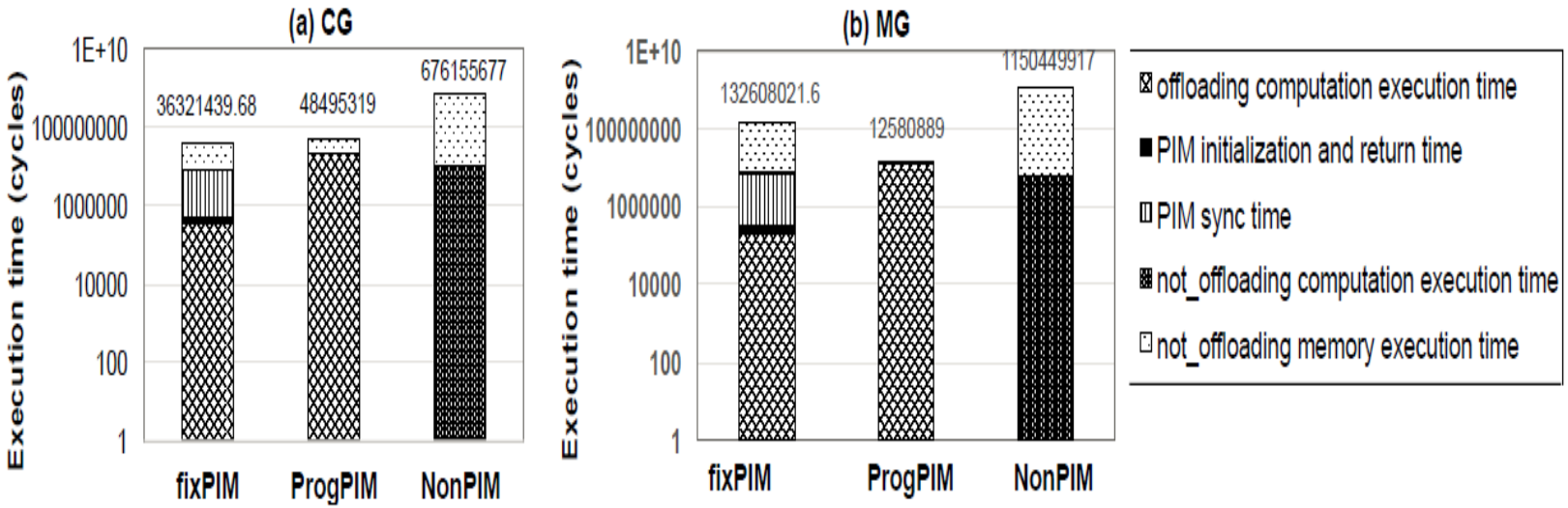


Figure 6: Preliminary results for two NPB benchmarks with emulation of fixed-function and programmable PIM.

- PIMs bring significant performance improvement:
  - CG: there are 18.6x (fixed-functional PIM) and 13.9x (programmable PIM) performance improvement over the non-PIM case
  - MG: there are 8.7x (fixed-functional PIM) and 91.4 (programmable PIM) performance improvement over the non-PIM case

Results: CG and MG

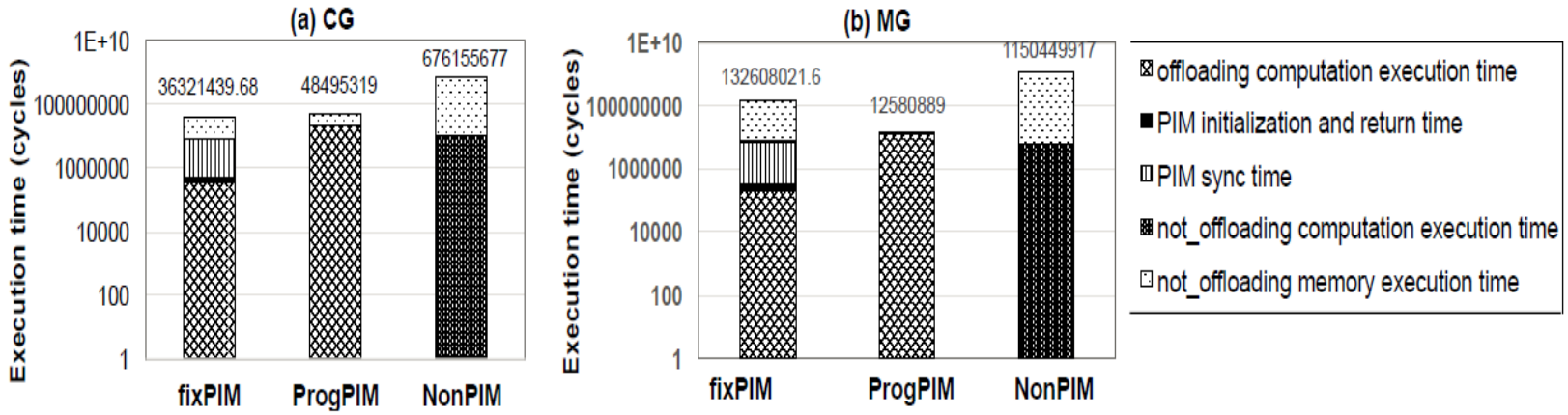


Figure 6: Preliminary results for two NPB benchmarks with emulation of fixed-function and programmable PIM.

- For CG, the fixed-function PIMs perform 90% better than the programmable PIM
  - The workloads offloaded to the fixed functional PIMs and programmable PIM are almost the same
  - Fixed-functional PIM provides more computation parallelism than the programmable PIM
- For MG, the programmable PIM performs 25% better than the fixed-functional PIMs
  - 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 limit the offloadability of workloads
  - The fixed-functional PIMs suffer form high PIM synchronization overhead (17% of the total execution time)

Results: GLADE

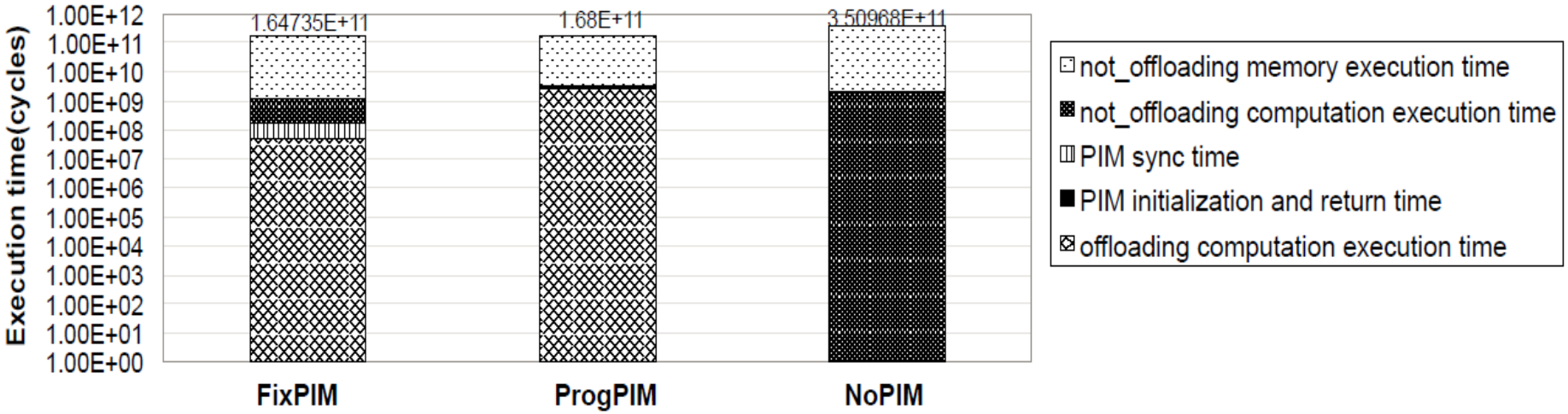


Figure 7: Preliminary results for GLADE with emulation of fixed-function and programmable PIM.

- PIMs bring significant performance improvement:
  - We have 2.13x and 2.09x speedup with the fixed-functional PIMs and programmable PIM respectively
- The fixed functional PIMs perform better than the programmable PIM (2% improvement)
  - The fixed-functional PIMs perform better because of massive numbers of operators, hence providing larger computation parallelism

Conclusions

- We review the limitations and strengths of the two PIMs
  - Computation parallelism
  - Synchronization overhead
  - Flexibility for workload offloading

- Neither the fixed-functional PIM nor programmable PIM can perform optimally in all cases