Providing Plug N’ Play for Processing-in-Memory Accelerators

Paulo C. Santos, Bruno E. Forlin (speaker), Luigi Carro
WHY PIM?

- **Main Memory**: 25.6GB/s per channel
- **Cache Level 3**: ≈ 76GB/s
- **Cache Level 2**: ≈ 120GB/s
- **Cache Level 1**: ≈ 248GB/s
- **Register File**: ≈ 1TB/s

Memory Size

Bandwidth
WHY PIM?

- **Main Memory**: 25.6GB/s / channel
  - **Cache Level 3**: ≈ 76GB/s
  - **Cache Level 2**: ≈ 120GB/s
  - **Cache Level 1**: ≈ 248GB/s
  - **Register File**: ≈ 1TB/s

- **Memory Size**

- **Bandwidth**

- **8Bytes 1 channel**
- **64Bytes 1 port**
- **64Bytes 1 port**
- **64Bytes 2 ports**
- **AVX-512bit registers**
WHY PIM?

- **Main Memory**
  - Bandwidth: 25.6GB/s / channel

- **Cache Level 3**
  - Bandwidth: ≈ 76GB/s

- **Cache Level 2**
  - Bandwidth: ≈ 120GB/s

- **Cache Level 1**
  - Bandwidth: ≈ 248GB/s

- **Register File**
  - Bandwidth: ≈ 1TB/s

Memory Size:
- **Main Memory**
  - Bandwidth: 8k Bytes

- **Cache L3**
- **Cache L2**
- **Cache L1**
- **Core**

Total Bandwidth: 64 Bytes
WHY PIM?

- **Main Memory**: 25.6GB/s / channel
- **Cache Level 3**: ≈ 76GB/s
- **Cache Level 2**: ≈ 120GB/s
- **Cache Level 1**: ≈ 248GB/s
- **Register File**: ≈ 1TB/s

Diagram showing:
- **Main Memory** with a bandwidth of 8k Bytes
- Cache levels: L3, L2, L1, Core
- Bandwidth: 64 Bits narrow bus
- Memory Size: 64 Bytes
WHY PIM?

Main Memory
- 100’s GB/s

Cache Level 3
- ≈ 76GB/s

Cache Level 2
- ≈ 120GB/s

Cache Level 1
- ≈ 248GB/s

Register File
- ≈ 1TB/s

Main Memory
- 8k Bytes

Cache L3

Cache L2

Cache L1

Core
- 64 Bytes

Bandwidth

Memory Size

64bits narrow bus
WHY PIM?

- Bandwidth
  - Main Memory: 100's GB/s
    - Cache Level 3: ≈ 76GB/s
      - Cache Level 2: ≈ 120GB/s
        - Cache Level 1: ≈ 248GB/s
          - Register File: ≈ 1TB/s

- Memory Size
  - Main Memory + PIM Units
    - 8k Bytes
      - 64bits narrow bus
        - Cache L3
          - Cache L2
            - Cache L1
              - Core
                - 64 Bytes
Interest in PIM

Energy Efficiency

Deep learning
Interest in PIM

- Energy Efficiency
- Big Data
- Pointer Chasing
- Pattern Matching
- Deep learning
- Graph applications
- Biological Neural Simulation
Interest in PIM

- Energy Efficiency
- Deep learning
- Graph applications
- Biological Neural Simulation
- Big Data
- Pointer Chasing
- Pattern Matching

No large scale adoption… Why?
Software Stack

- Code
- Library
- Compiler
- Driver
- OS
Software Stack

C
C++

Code

Library

Compiler

Driver

OS
Software Stack

H. Ahmed et al., "A Compiler for Automatic Selection of Suitable Processing-in-Memory Instructions," DATE, 2019
Software Stack

- C
- C++
- Code
- Library
- Compiler
- Driver
- OS
- Functions
- Intrinsics
- CUDA
- PRIMO
- Ahmed et al., 2019
- Device Management

H. Ahmed et al., "A Compiler for Automatic Selection of Suitable Processing-in-Memory Instructions," DATE, 2019
Hardware Support

Memory Module

DRAM  DRAM  DRAM  DRAM

PIM

Processor

Memory Controller
Cache Hierarchy
TLB
Core
Hardware Support

Memory Module

DRAM
DRAM
DRAM
DRAM

PIM

Memory Controller
Cache Hierarchy
TLB
Core

ISA Extensions
Hardware Support

- Memory Controller
- Cache Hierarchy
- TLB
- Core
- PIM
- Additional Blocks
- ISA Extensions
Memory Controller

Cache Hierarchy

DRAM

DRAM

DRAM

DRAM

PIM

Processor

Memory Controller

Cache Hierarchy

TLB

Core

Modifications to the Caches

Additional Blocks

ISA Extensions
Hardware Support

Memory Controller

Cache Hierarchy

TLB

Core

 Modifications to the MC

 Modifications to the Caches

 Additional Blocks

 ISA Extensions
1. MC selects PIM and memory targets
2. Caches filter addresses and handle coherence
3. Additional HW filters memory access
4. Core can decode PIM instructions
Design constraints:
- Increased design costs
- Increased Area and Power
Hardware Support

Design constraints:
- Increased design costs
- Increased Area and Power

Barrier of Entry:
- Limits compatibility
- Hard to experiment with new PIM designs and software
We want an integration solution that

Fits any host
Fits any accelerator
Can be automated

This means

No HW modification at host
Full General Purpose compliance
No programmer burden

How do we make the PIM Plug&Play?
Outline

• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
Target PIM - DDR

Typical DDR

DRAM

DRAM

DRAM

DRAM

64bits
to/from CPU
Target PIM - DDR

Target PIM - DDR


Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. MICRO '52, 2019

V. Seshadri et al., 2017

Gao et al., 2019
3D-stacked DRAM Layers

Long Links to/from CPU
Target PIM - 3D Stacked

3D-stacked DRAM Layers

Processing-in-Memory within logic layer

Long Links to/from CPU

Santos et al., 2017

Target PIM - 3D Stacked

3D-stacked DRAM Layers

Long Links to/from CPU

Processing-in-Memory within logic layer

In-Memory Processing close to DRAM cells

Santos et al., 2017

Gao et al., 2019


Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. MICRO '52, 2019
Target PIM - Memristor Array

Memristor Array

DAC

ADC

Row Buffer - 8kB
Target PIM - Memristor Array

Memristor Array

DAC

ADC

Row Buffer - 8kB

MEMRISYS, 2015

Vector Operation

Hamdioui et al., 2015
Target PIM - Common Problems

Memory Controller
Cache Hierarchy
TLB
Core
DAC
ADC
Row Buffer - 8kB
Outline

• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
Outline

• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
Plug N’ PIM – Memory Mapped PIM

• Plug N’ PIM relies on *in-memory mapped* approach
  • An idea from the 1970s
  • 8085 architecture
Memory Controller

Memory address selector

I/O address selector

Memory Bus

Cache Hierarchy

TLB

Core
Plug N’ PIM – Memory Mapped PIM
Plug N’ PIM – Memory Mapped PIM

Memory Module

DRAM
PIM
0x0000

DRAM
PIM
0x0004

DRAM
PIM
0x0008

DRAM
PIM
0x000C

Instruction Manager

Memory Controller

Cache Hierarchy

TLB

Core

PIM Instruction to PIM 0x0000
• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
Plug N’ PIM – Code Offloading
Plug N’ PIM – Code Offloading

PIM EXPLICIT ASSEMBLY CODE

\[
\text{mov rax, -16384} \\
\text{PIM\_8192B\_LOAD\_DWORD VPU\_R\_0, [rax+b+16384]} \\
\text{add rcx, 5}
\]
Plug N’ PIM – Code Offloading

PIM EXPLICIT ASSEMBLY CODE

```
mov rax, -16384
PIM_8192B_LOAD_DWORD VPU_R_0, [rax+b+16384]
add rcx, 5
```
• Mixed host + PIM code

C code

```c
for (int i = 0; i < 3136; i++)
    int_a[i] = int_MEM[i];
```

PIM EXPLICIT ASSEMBLY CODE

```asm
mov rax, -16384
PIM_8192B_LOAD_DWORD VPU_R_0, [rax+b+16384]
........
add rcx, 5
........
vmodqu xmm2, xmm1, [rcx+382]
```
• But how can the host decode these instructions?
• But how can the host decode these instructions?

• Simple! It does not!
But how can the host decode these instructions?

Simple! It does not

PIM instruction offloaded as DATA
Plug N’ PIM – Code Offloading

• But how can the host decode these instructions?

• Simple! It does not

• PIM instruction offloaded as DATA

• Via non-temporal $STORE$ instructions
Plug N’ PIM – Code Offloading

• Extra Work for the compiler

PIM EXPLICIT ASSEMBLY CODE

```markdown
mov rax, -16384
PIM 8192B LOAD DWORD VPU R 0,[rax+b+16384]

........
add rcx, 5
........

vmodqu xmm2, [rcx+382]
```
Extra Work for the compiler

**PIM EXPLICIT ASSEMBLY CODE**

```assembly
mov rax, -16384
PIM 8192B LOAD DWORD VPU R 0,[rax+b+16384]
........
add rcx, 5
........
vmodqu xmm2, [rcx+382]
```

Compiler Translates PIM Instruction to ordinary **DATA**
• Extra Work for the compiler

PIM EXPLICIT ASSEMBLY CODE

```asm
mov rax, -16384
PIM 8192B LOAD DWORD VPU R 0, [rax+b+16384]
........
add rcx, 5
........
vmodqu xmm2, [rcx+382]
```

Compiler Translates PIM Instruction to ordinary DATA

Instruction emitted via non-temporal Store

```asm
movntd $DATA, [PIM_ADDRESS]
```
Plug N’ PIM – Code Offloading

• Extra Work for the compiler

PIM EXPLICIT ASSEMBLY CODE

```assembly
mov rax, -16384
PIM 8192B LOAD DWORD VPU R 0, [rax+b+16384]
........
add rcx, 5
........
vmodqu xmm2, [rcx+382]
```

Compiler Translates
PIM Instruction to ordinary **DATA**

Instruction emitted via non-temporal Store

```assembly
movntd $DATA, [PIM_ADDRESS]
```

PIM INSTRUCTION EMISSION
BYPASSES THE CACHE HIERARCHY
Plug N’ PIM – Code Offloading

Typical NON-TEMPORAL STORE instruction behavior
Outline

• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
Plug N’ PIM – Cache Coherence

Memory Module

DRAM
PIM
0x0000
DRAM
PIM
0x0004
DRAM
PIM
0x0008
DRAM
PIM
0x000C

Instruction Manager

Processor

Memory Controller

Cache Hierarchy

TLB

Core
Plug N’ PIM – Cache Coherence

[Diagram showing a memory module with DRAM and PIM components, connected to an Instruction Manager, Memory Controller, Cache Hierarchy, TLB, and Core.]
Plug N’ PIM – Cache Coherence
Plug N’ PIM – Cache Coherence

Memory Module

Instruction Manager

Memory Controller

Cache Hierarchy

TLB

Core

DRAM
PIM
0x0000

DRAM
PIM
0x0004

DRAM
PIM
0x0008

DRAM
PIM
0x000C
Plug N’ PIM – Cache Coherence
• Cache conflicts between host and PIM

```
PIM EXPLICIT ASSEMBLY CODE

........

vmodqu xmm2, [rax+b+16384]
PIM_LOAD  VPU_R_0,[rax+b+16384]
........
```
• Cache conflicts between host and PIM

```
.....
vmodqu xmm2, [rax+b+16384]
PIM_LOAD VPU_R_0,[rax+b+16384]
.....
```

MEMORY ADDRESS PREVIOUSLY ACCESSED BY THE HOST
Plug N’ PIM – Cache Coherence

• Supported by cache line FLUSH and FENCE instructions

PIM EXPLICIT ASSEMBLY CODE

........
vmodqu xmm2, [rax+b+16384]
PIM_LOAD VPU_R_0, [rax+b+16384]
........
Plug N’ PIM – Cache Coherence

• Supported by cache line FLUSH and FENCE instructions

```
PIM EXPLICIT ASSEMBLY CODE

......
vmodqu xmm2, [rax+b+16384]
PIM_LOAD VPU_R_0, [rax+b+16384]
......
```

Compiler finds “dirty” cache position
• Supported by cache line FLUSH and FENCE instructions
Plug N’ PIM – Cache Coherence

- Supported by cache line FLUSH and FENCE instructions

PIM EXPLICIT ASSEMBLY CODE

........

vmodqu xmm2, [rax+b+16384]
PIM_LOAD VPU_R_0, [rax+b+16384]
........

Compiler finds “dirty” cache position

Handled by cache line FLUSH and FENCE Instructions

- FLUSH + FENCE added before NT store
  - clflush 0[rax+b+16384]
  - mfence
  - movntdq $DATA, [PIM_ADDRESS]
Plug N’ PIM – Cache Coherence

Memory Module
- DRAM
  - PIM 0x0000
  - PIM 0x0004
  - PIM 0x0008
  - PIM 0x000C

Instruction Manager

Memory Controller

Cache Hierarchy

TLB

Core
Plug N’ PIM – Cache Coherence

FLUSH instruction writes-back the selected cache lines
• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
Plug N’ PIM – Virtual Memory Management
• Two host non temporal stores are adopted
• Two host non temporal stores are adopted
Plug N’ PIM – Virtual Memory Management

• Two host non temporal stores are adopted
Plug N’ PIM – Virtual Memory Management

• Two host non temporal stores are adopted

```
PIM_LOAD VPU_R_0, [rax+b+16384]
```

```
movntdq $DATA, [PIM_ADDRESS]
```

Virtual Address
Plug N’ PIM – Virtual Memory Management

• Two host non temporal stores are adopted

PIM_LOAD VPU_R_0, [rax+b+16384]

movntdq $DATA, [PIM_ADDRESS]
• Two host non temporal stores are adopted
Two host non temporal stores are adopted
Plug N’ PIM – Virtual Memory Management

Instruction Manager

Memory Module

DRAM
PIM
0x0000

DRAM
PIM
0x0004

DRAM
PIM
0x0008

DRAM
PIM
0x000C

Memory Controller
Cache Hierarchy
TLB
Core
Plug N’ PIM – Virtual Memory Management

1. PIM Instruction is emitted
1. PIM Instruction is emitted
2. **Instruction Manager** receives PIM Instruction if LOAD/STORE captures next instruction if DATA == previous PIM Instruction gets physical address
1. PIM Instruction is emitted

2. **Instruction Manager** receives PIM Instruction if LOAD/STORE captures next instruction if DATA == previous PIM Instruction gets physical address

3. **Instruction Manager** reconstructs PIM Instruction with physical address
Outline

• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
## Evaluation

<table>
<thead>
<tr>
<th>Sim²PIM Simulation environment - Santos et al., 2021</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Baseline and Host Processors</strong></td>
</tr>
<tr>
<td>**Intel i7-860@ 2.8GHz</td>
</tr>
<tr>
<td><strong>Target PIM</strong></td>
</tr>
<tr>
<td><strong>RVU - Santos et al., 2017</strong></td>
</tr>
<tr>
<td>1GHz; 32 Independent Functional Units; Int. and FP Capable;</td>
</tr>
<tr>
<td>Instructions from 128Bytes to 8192Bytes;</td>
</tr>
<tr>
<td>Vector Operations up to 256Bytes per Vector Processing Unit;</td>
</tr>
<tr>
<td>32 Independent Register Bank of 8x256Bytes (one per VPU/vault);</td>
</tr>
<tr>
<td>Supported by 3D-Stacked memory inspired by HMC</td>
</tr>
<tr>
<td><strong>Compiler</strong></td>
</tr>
<tr>
<td><strong>PRIMO - Ahmed et al., 2019</strong></td>
</tr>
</tbody>
</table>

P. C. Santos et al., “Operand size reconfiguration for big data processing in memory,” 2017, DATE
Evaluation

- We evaluated two simple kernels
  - Vecsum
  - DotProduct
Evaluation

• We evaluated two simple kernels
  • Vecsum
  • DotProduct

• And a widely exploited CNN application
  • Yolo V3
Evaluation

• We evaluated two simple kernels
  • Vecsum
  • DotProduct

• And a widely exploited CNN application
  • Yolo V3

• 3 data localities
  • 256B
  • 1KB
  • 8KB
Evaluation – Code Offloading Impact

VECSUM 8MB

Cycles - Normalized

RVU 256B

i7-860  i7-4700  R5-1600

computing  offloading  coherence
Evaluation – Code Offloading Impact

[Chart showing cycles normalized for different processors: i7-860, i7-4700, R5-1600. The chart highlights RVU 256B with computing, offloading, and coherence components.]
Evaluation – Code Offloading Impact

VECSUM 8MB

Cycles - Normalized

i7-860  i7-4700  R5-1600

RVU 256B

computing  offloading  coherence
Evaluation – Code Offloading Impact

VECSUM 8MB

<table>
<thead>
<tr>
<th>Cycles - Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>0.9</td>
</tr>
<tr>
<td>0.8</td>
</tr>
<tr>
<td>0.7</td>
</tr>
<tr>
<td>0.6</td>
</tr>
<tr>
<td>0.5</td>
</tr>
<tr>
<td>0.4</td>
</tr>
<tr>
<td>0.3</td>
</tr>
<tr>
<td>0.2</td>
</tr>
<tr>
<td>0.1</td>
</tr>
</tbody>
</table>

i7-860  i7-4700  R5-1600  i7-860  i7-4700  R5-1600

RVU 256B

RVU 1024B

- computing
- offloading
- coherence
Evaluation – Code Offloading Impact

VECSUM 8MB

Cycles - Normalized

RVU 256B

i7-860  i7-4700  R5-1600

RVU 1024B

i7-860  i7-4700  R5-1600

computing  offloading  coherence
Evaluation – Code Offloading Impact

VECSUM 8MB

Cycles - Normalized

<table>
<thead>
<tr>
<th></th>
<th>i7-860</th>
<th>i7-4700</th>
<th>R5-1600</th>
<th>i7-860</th>
<th>i7-4700</th>
<th>R5-1600</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVU 256B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RVU 1024B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- blue: computing
- orange: offloading
- gray: coherence
Evaluation – Code Offloading Impact

VECSUM 8MB

Cycles - Normalized

0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

i7-860 i7-4700 R5-1600 i7-860 i7-4700 R5-1600 i7-860 i7-4700 R5-1600

RVU 256B RVU 1024B RVU 8192B

computing offloading coherence
Evaluation – Code Offloading Impact

VECSUM 8MB

Cycles - Normalized

<table>
<thead>
<tr>
<th></th>
<th>RVU 256B</th>
<th>RVU 1024B</th>
<th>RVU 8192B</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-860</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i7-4700</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R5-1600</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Computing
- Offloading
- Coherence

RVU 8192B
Evaluation – Code Offloading Impact

**VECSUM 8MB**

Higher DLP per instruction = smaller impact of code offloading
Evaluation – Cache Coherence Impact

DOTPRODUCT 8MB
DOTPRODUCT 8MB

Evaluation – Cache Coherence Impact
Half of the cycles are used to keep the cache coherent.
Evaluation – Cache Coherence Impact

DOTPRODUCT 8MB

<table>
<thead>
<tr>
<th>CPU</th>
<th>RVU</th>
<th>Cycles - Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-860</td>
<td>256B</td>
<td>computing (0.4)</td>
</tr>
<tr>
<td>i7-4700</td>
<td></td>
<td>offloading (0.1)</td>
</tr>
<tr>
<td>R5-1600</td>
<td></td>
<td>coherence (0.5)</td>
</tr>
</tbody>
</table>

- RVU 256B
- RVU 1024B
- RVU 8192B

Legend:
- blue: computing
- orange: offloading
- gray: coherence
In case of larger SIMD operations, nearly 95% of the time is spent for cache coherence.
Evaluate only the impact of cache coherence
Evaluation – Cache Coherence Impact

VECSUM 8MB

- no Cache Coherence
- All Addresses Cache Coherence

Speedup over SSE Cycles - Normalized

- i7-860
- i7-4700
- R5-1600

RVU 256B
Evaluation – Cache Coherence Impact

VECSUM 8MB

- no Cache Coherence
- All Addresses Cache Coherence

Cycles - Normalized

Speedup over SSE

i7-860  i7-4700  R5-1600  i7-860  i7-4700  R5-1600

RVU 256B  RVU 1024B
Evaluation – Cache Coherence Impact

VECSUM 8MB

- **No Cache Coherence**
- **All Addresses Cache Coherence**

<table>
<thead>
<tr>
<th>Speedup over SSE Cycles - Normalized</th>
<th>i7-860</th>
<th>i7-4700</th>
<th>R5-1600</th>
<th>i7-860</th>
<th>i7-4700</th>
<th>R5-1600</th>
<th>i7-860</th>
<th>i7-4700</th>
<th>R5-1600</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVU 256B</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RVU 1024B</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>RVU 8192B</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>
Evaluation – Cache Coherence Impact

VECSUM 8MB

- No Cache Coherence
- All Addresses Cache Coherence

Speedup over SSE Cycles - Normalized

i7-860  i7-4700  R5-1600  i7-860  i7-4700  R5-1600  i7-860  i7-4700  R5-1600

RVU 256B  RVU 1024B  RVU 8192B
VECSUM 8MB

Huge impact if all addresses are flushed

Only need to flush previously modified address!
Evaluation – Cache Coherence Impact

Convolutional Neural Network (Yolov3)
Evaluation – Cache Coherence Impact

Convolutional Neural Network (Yolov3)

- i7-860+PIM
- i7-4700+PIM
- R5-1600+PIM

Speedup over SSE
Cycles - Normalized

Flush every access
Evaluation – Cache Coherence Impact

Convolutional Neural Network (Yolov3)

- max speedup 1.8x
Convolutional Neural Network (Yolov3)

- i7-860+PIM
- i7-4700+PIM
- R5-1600+PIM

Evaluation – Cache Coherence Impact

Speedup over SSE Cycles - Normalized

Flush every access

No-Coherence

118
Convolutional Neural Network (Yolov3)

- i7-860+PIM
- i7-4700+PIM
- R5-1600+PIM

Speedup over SSE Cycles - Normalized

- Flush every access
- No-Coherence

up to 5.2x speedup
Evaluation – Cache Coherence Impact

Convolutional Neural Network (Yolov3)

Keeping coherence of previously modified address, up to 4x
Evaluation – Cache Coherence Impact

Convolutional Neural Network (Yolov3)

Keeping coherence of previously modified address, up to 4x
• Target PIM

• Plug N’ PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
Outline

• Target PIM

• Plug N' PIM
  • Code Offloading
  • Cache Memory Coherence
  • Virtual Memory Management

• Evaluation

• Conclusion and Future Work
Conclusion and Future Work

• Code Offloading + Cache Coherence + Virtual Memory support
  • No hardware overhead at host side
  • Minimal design effort at PIM
  • Full compatibility with current processors
  • Provides significant speedups

• Future Work
  • Improve coherence mechanisms
Thank you!

Providing Plug N’ Play for Processing-in-Memory Accelerators

Paulo C. Santos, Bruno E. Forlin (speaker), Luigi Carro

bruno.eforlin@inf.ufrgs.br

For more information on this project and others, go to: https://pim.computer/