Tutorial on Memory-Centric Computing:
PIM Adoption & Programmability

Geraldo F. Oliveira
Prof. Onur Mutlu

ISCA 2024
29 June 2024
Agenda

- Introduction to Memory-Centric Computing Systems
- Invited Talk by Prof. Minsoo Rhu: “Memory-Centric Computing Systems – For AI and Beyond”
- Coffee Break
- Real-World Processing-Near-Memory Systems
- Invited Talk by Prof. Saugata Ghose: “RACER and ReRAM PUM”
- PIM Programming & Infrastructure for PIM Research
- Closing Remarks
Processing in Memory: Adoption Challenges

1. Processing near Memory
2. Processing using Memory
Eliminating the Adoption Barriers

How to Enable Adoption of Processing in Memory
Potential Barriers to Adoption of PIM

1. **Applications & software** for PIM

2. Ease of **programming** (interfaces and compiler/HW support)

3. **System** and **security** support: coherence, synchronization, virtual memory, isolation, communication interfaces, ...

4. **Runtime** and **compilation** systems for adaptive scheduling, data mapping, access/sharing control, ...

5. **Infrastructures** to assess benefits and feasibility

All can be solved with change of mindset
We Need to Revisit the Entire Stack

We can get there step by step
Adoption: How to Keep It Simple?

[Slides (pdf)] [Lightning Session Slides (pdf)]

PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture

Junwhan Ahn  Sungjoo Yoo  Onur Mutlu†  Kiyoungh Choi
junwhan@snu.ac.kr, sungjoo.yoo@gmail.com, onur@cmu.edu, kchoi@snu.ac.kr

Seoul National University  †Carnegie Mellon University
Adoption: How to Ease Programmability? (I)

- Geraldo F. Oliveira, Alain Kohli, David Novo, Juan Gómez-Luna, Onur Mutlu,
  “DaPPA: A Data-Parallel Framework for Processing-in-Memory Architectures,”
in PACT SRC Student Competition, Vienna, Austria, October 2023.

DaPPA: A Data-Parallel Framework for Processing-in-Memory Architectures

Geraldo F. Oliveira* Alain Kohli* David Novo‡ Juan Gómez-Luna* Onur Mutlu*
*ETH Zürich ‡LIRMM, Univ. Montpellier, CNRS
Jinfan Chen, Juan Gómez-Luna, Izzat El Hajj, YuXin Guo, and Onur Mutlu,
"SimplePIM: A Software Framework for Productive and Efficient Processing in Memory"
Proceedings of the 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria, October 2023.

SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory

Jinfan Chen¹  Juan Gómez-Luna¹  Izzat El Hajj²  Yuxin Guo¹  Onur Mutlu¹
¹ETH Zürich  ²American University of Beirut
SimplePIM:
A Software Framework for Productive and Efficient Processing-in-Memory

Jinfan Chen, Juan Gómez Luna, Izzat El Hajj, Yuxin Guo, Onur Mutlu

https://github.com/CMU-SAFARI/SimplePIM
Executive Summary

• Processing-in-Memory (PIM) promises to alleviate the *data movement bottleneck*

• Real PIM hardware is now available, e.g., UPMEM PIM

• However, *programming real PIM hardware is challenging*, e.g.:
  - Distribute data across PIM memory banks,
  - Manage data transfers between host cores and PIM cores, and between PIM cores,
  - Launch PIM kernels on the PIM cores, etc.

• *SimplePIM* is a high-level programming framework for real PIM hardware
  - Iterators such as `map`, `reduce`, and `zip`
  - Collective communication with `broadcast`, `scatter`, and `gather`

• Implementation on UPMEM and evaluation with six different workloads
  - Reduction, vector add, histogram, linear/logistic regression, K-means
  - **4.4x fewer lines of code** compared to hand-optimized code
  - Between 15% and 43% **faster than hand-optimized code** for three workloads

• Source code: [https://github.com/CMU-SAFARI/SimplePIM](https://github.com/CMU-SAFARI/SimplePIM)
Outline

- Processing-in-memory and PIM programming
- SimplePIM: A high-level programming framework for processing-in-memory
- Evaluation
Processing-in-Memory (PIM)

- PIM is a computing paradigm that advocates for memory-centric computing systems, where processing elements are placed near or inside the memory arrays.

- Real-world PIM architectures are becoming a reality:
  - UPMEM PIM, Samsung HBM-PIM, Samsung AxDIMM, SK Hynix AiM, Alibaba HB-PNM

- These PIM systems have some common characteristics:
  1. There is a host processor (CPU or GPU) with access to (1) standard main memory, and (2) PIM-enabled memory.
  2. PIM-enabled memory contains multiple PIM processing elements (PEs) with high bandwidth and low latency memory access.
  3. PIM PEs run only at a few hundred MHz and have a small number of registers and small (or no) cache/scratchpad.
  4. PIM PEs may need to communicate via the host processor.
A State-of-the-Art PIM System

• In our work, we use the UPMEM PIM architecture
  - General-purpose processing cores called DRAM Processing Units (DPUs)
    • Up to 24 PIM threads, called tasklets
    • 32-bit integer arithmetic, but multiplication/division are emulated*, as well as floating-point operations
  - 64-MB DRAM bank (MRAM), 64-KB scratchpad (WRAM)

* 8-bit integer multiplication is natively supported
Programming a PIM System (I)

• Example: Hand-optimized histogram with UPMEM SDK

```c
... // Initialize global variables and functions for histogram
int main_kernel() {
    if (tasklet_id == 0)
        mem_reset(); // Reset the heap
    ... // Initialize variables and the histogram
    T *input_buff_A = (T*)mem_alloc(2048); // Allocate buffer in scratchpad memory

    for (unsigned int byte_index = base_tasklet; byte_index < input_size; byte_index += stride) {
        // Boundary checking
        uint32_t l_size_bytes = (byte_index + 2048 >= input_size) ? (input_size - byte_index) : 2048;
        // Load scratchpad with a DRAM block
        mram_read((const __mram_ptr void*)(mram_base_addr_A + byte_index), input_buff_A, l_size_bytes);
        // Histogram calculation
        histogram(hist, bins, input_buff_A, l_size_bytes/sizeof(uint32_t));
    }
    barrier_wait(&my_barrier); // Barrier to synchronize PIM threads
    ... // Merging histograms from different tasklets into one histo_dpu
    // Write result from scratchpad to DRAM
    if (tasklet_id == 0)
        if (bins * sizeof(uint32_t) <= 2048)
            mram_write(histo_dpu, (__mram_ptr void*)mram_base_addr_histo, bins * sizeof(uint32_t));
        else
            for (unsigned int offset = 0; offset < ((bins * sizeof(uint32_t)) >> 11); offset++) {
                mram_write(histo_dpu + (offset << 9), (__mram_ptr void*)(mram_base_addr_histo + (offset << 11)), 2048);
            }
    return 0;
}
```
Programming a PIM System (II)

- PIM programming is challenging
  - Manage data movement between host DRAM and PIM DRAM
    - Parallel, serial, broadcast, and gather/scatter transfers
  - Manage data movement between PIM DRAM bank and scratchpad
    - 8-byte aligned and maximum of 2,048 bytes
  - Multithreaded programming model
  - Inter-thread synchronization
    - Barriers, handshakes, mutexes, and semaphores

Our Goal
Design a high-level programming framework that abstracts these hardware-specific complexities and provides a clean yet powerful interface for ease of use and high program performance
Outline

Processing-in-memory and PIM programming

SimplePIM:
A high-level programming framework for processing-in-memory

Evaluation
The SimplePIM Programming Framework

• SimplePIM provides standard abstractions to build and deploy applications on PIM systems

  - Management interface
    • Metadata for PIM-resident arrays

  - Communication interface
    • Abstractions for host-PIM and PIM-PIM communication

  - Processing interface
    • Iterators (map, reduce, zip) to implement workloads
Management Interface

• Metadata for PIM-resident arrays
  - array_meta_data_t describes a PIM-resident array
  - simple_pim_management_t for managing PIM-resident arrays

• lookup: Retrieves all relevant information of an array

```c
array_meta_data_t* simple_pim_array_lookup(const char* id,
simple_pim_management_t* management);
```

• register: Registers the metadata of an array

```c
void simple_pim_array_register(array_meta_data_t* meta_data,
simple_pim_management_t* management);
```

• free: Removes the metadata of an array

```c
void simple_pim_array_free(const char* id, simple_pim_management_t* management);
```
The SimplePIM Programming Framework

- SimplePIM provides standard abstractions to build and deploy applications on PIM systems
  - Management interface
    - Metadata for PIM-resident arrays
  - Communication interface
    - Abstractions for host-PIM and PIM-PIM communication
  - Processing interface
    - Iterators (map, reduce, zip) to implement workloads
Host-to-PIM Communication: Broadcast

• SimplePIM Broadcast
  - Transfers a host array to all PIM cores in the system

```c
void simple_pim_array_broadcast(char* const id, void* arr, uint64_t len,
uint32_t type_size, simple_pim_management_t* management);
```
Host-to-PIM Communication: Scatter/Gather

- **SimplePIM Scatter**
  - Distributes an array to PIM DRAM banks

```c
void simple_pim_array_scatter(const char* id, void* arr, uint64_t len, uint32_t type_size, simple_pim_management_t* management);
```

- **SimplePIM Gather**
  - Collects portions of an array from PIM DRAM banks

```c
void* simple_pim_array_gather(const char* id, simple_pim_management_t* management);
```
PIM-PIM Communication: AllReduce

• SimplePIM AllReduce
  - Used for algorithm synchronization
  - The programmer specifies an accumulative function

```c
void simple_pim_array_allreduce(char* const id, handle_t* handle,
                                 simple_pim_management_t* management);
```

Before PIM-PIM communication

After PIM-PIM communication

SimplePIM AllReduce
PIM-PIM Communication: AllGather

• SimplePIM AllGather
  - Combines array pieces and distributes the complete array to all PIM cores

```c
void simple_pim_array_allgather(char* const id, char* new_id, simple_pim_management_t* management);
```
The SimplePIM Programming Framework

- SimplePIM provides standard abstractions to build and deploy applications on PIM systems
  - Management interface
    - Metadata for PIM-resident arrays
  - Communication interface
    - Abstractions for host-PIM and PIM-PIM communication
  - Processing interface
    - Iterators (map, reduce, zip) to implement workloads
Processing Interface: Map

• Array Map
  - Applies `map_func` to every element of the data array

```c
void simple_pim_array_map(const char* src_id, const char* dest_id,
uint32_t output_type, handle_t* handle, simple_pim_management_t* management);
```
Processing Interface: Reduction

• Array Reduction
  - The `map_to_val_func` function transforms an input element to an output value and an output index
  - The `acc_func` function accumulates the output values onto the output array

```c
void simple_pim_array_red(const char* src_id, const char* dest_id, uint32_t output_type, uint32_t output_len, handle_t* handle, simple_pim_management_t* management);
```
Processing Interface: Zip

• Array Zip
  - Takes two input arrays and combines their elements into an output array

```c
void simple_pim_array_zip(const char* src1_id, const char* src2_id, const char* dest_id, simple_pim_management_t* management);
```
General Code Optimizations

• Strength reduction
• Loop unrolling
• Avoiding boundary checks
• Function inlining
• Adjustment of data transfer sizes
More in the Paper

• Strength reduction

• Loop unrolling

**SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory**

Jinfan Chen\(^1\)  Juan Gómez-Luna\(^1\)  Izzat El Hajj\(^2\)  Yuxin Guo\(^1\)  Onur Mutlu\(^1\)

\(^1\)ETH Zürich  \(^2\)American University of Beirut

• Function inlining

• Adjunction

Outline

Processing-in-memory and PIM programming

SimplePIM: A high-level programming framework for processing-in-memory

Evaluation
Evaluation Methodology

• Evaluated system
  - UPMEM PIM system with 2,432 PIM cores with 159 GB of PIM DRAM

• Real-world Benchmarks
  - Vector addition
  - Reduction
  - Histogram
  - K-Means
  - Linear regression
  - Logistic regression

• Comparison to hand-optimized codes in terms of programming productivity and performance
Productivity Improvement (I)

- Example: Hand-optimized histogram with UPMEM SDK

```c
... // Initialize global variables and functions for histogram
int main_kernel() {
  if (tasklet_id == 0)
    mem_reset(); // Reset the heap
  ... // Initialize variables and the histogram
  T *input_buff_A = (T*)mem_alloc(2048); // Allocate buffer in scratchpad memory

  for (unsigned int byte_index = base_tasklet; byte_index < input_size; byte_index += stride) {
    // Boundary checking
    uint32_t l_size_bytes = (byte_index + 2048 >= input_size) ? (input_size - byte_index) : 2048;
    // Load scratchpad with a DRAM block
    mram_read((const __mram_ptr void*)mram_base_addr_A + byte_index), input_buff_A, l_size_bytes);
    // Histogram calculation
    histogram(hist, bins, input_buff_A, l_size_bytes/sizeof(uint32_t));
  }
  barrier_wait(&my_barrier); // Barrier to synchronize PIM threads
  ... // Merging histograms from different tasklets into one histo_dpu
  // Write result from scratchpad to DRAM
  if (tasklet_id == 0)
    if (bins * sizeof(uint32_t) <= 2048)
      mram_write(histo_dpu, (__mram_ptr void*)mram_base_addr_histo, bins * sizeof(uint32_t));
    else
      for (unsigned int offset = 0; offset < ((bins * sizeof(uint32_t)) >> 11); offset++) {
        mram_write(histo_dpu + (offset << 9), (__mram_ptr void*)(mram_base_addr_histo +
          (offset << 11)), 2048);
      }
  return 0;
}
```
Productivity Improvement (II)

• Example: SimplePIM histogram

```c
// Programmer-defined functions in the file "histo_filepath"
void init_func (uint32_t size, void* ptr) {
    char* casted_value_ptr = (char*) ptr;
    for (int i = 0; i < size; i++)
        casted_value_ptr[i] = 0;
}

void acc_func (void* dest, void* src) {
    *(uint32_t*)dest += *(uint32_t*)src;
}

void map_to_val_func (void* input, void* output, uint32_t* key) {
    uint32_t d = *((uint32_t*)input);
    *(uint32_t*)output = 1;
    *key = d * bins >> 12;
}

// Host side handle creation and iterator call
handle_t* handle = simple_pim_create_handle("histo_filepath", REDUCE, NULL, 0);

// Transfer (scatter) data to PIM, register as "t1"
simple_pim_array_scatter("t1", src, bins, sizeof(T), management);

// Run histogram on "t1" and produce "t2"
simple_pim_array_red("t1", "t2", sizeof(T), bins, handle, management);
```
### Productivity Improvement (III)

- Lines of code (LoC) reduction

<table>
<thead>
<tr>
<th></th>
<th>SimplePIM</th>
<th>Hand-optimized</th>
<th>LoC Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction</td>
<td>14</td>
<td>83</td>
<td>5.93×</td>
</tr>
<tr>
<td>Vector Addition</td>
<td>14</td>
<td>82</td>
<td>5.86×</td>
</tr>
<tr>
<td>Histogram</td>
<td>21</td>
<td>114</td>
<td>5.43×</td>
</tr>
<tr>
<td>Linear Regression</td>
<td>48</td>
<td>157</td>
<td>3.27×</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>59</td>
<td>176</td>
<td>2.98×</td>
</tr>
<tr>
<td>K-Means</td>
<td>68</td>
<td>206</td>
<td>3.03×</td>
</tr>
</tbody>
</table>

SimplePIM reduces the number of lines of effective code by a factor of 2.98× to 5.93×
Performance Evaluation (I)

• Weak scaling analysis

SimplePIM achieves **comparable performance** for reduction, histogram, and linear regression.

SimplePIM **outperforms** hand-optimized implementations for vector addition, logistic regression, and k-means by 10%-37%.
Performance Evaluation (II)

- Strong scaling analysis

SimplePIM scales better than hand-optimized implementations for reduction, histogram, and linear regression.

SimplePIM outperforms hand-optimized implementations for vector addition, logistic regression, and $k$-means by 15%-43%. 
Discussion

• SimplePIM is devised for PIM architectures with
  - A host processor with access to standard main memory and PIM-enabled memory
  - PIM processing elements (PEs) that communicate via the host processor
  - The number of PIM PEs scales with memory capacity

• SimplePIM emulates the communication between PIM cores via the host processor

• Other parallel patterns can be incorporated in future work
  - Prefix sum and filter can be easily added
  - Stencil and convolution would require fine-grained scatter-gather for halo cells
  - Random access patterns would be hard to support
SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory

Jinfan Chen$^1$  Juan Gómez-Luna$^1$  Izzat El Hajj$^2$  Yuxin Guo$^1$  Onur Mutlu$^1$

$^1$ETH Zürich  $^2$American University of Beirut

Source Code

- [https://github.com/CMU-SAFARI/SimplePIM](https://github.com/CMU-SAFARI/SimplePIM)

**SimplePIM**

This project implements SimplePIM, a software framework for easy and efficient in-memory-hardware programming. The code is implemented on UPMEM, an actual, commercially available PIM hardware that combines traditional DRAM memory with general-purpose in-order cores inside the same chip. SimplePIM processes arrays of arbitrary elements on a PIM device by calling iterator functions from the host and provides primitives for communication among PIM cores and between PIM and the host system.

We implement six applications with SimplePIM on UPMEM:

- Vector Addition
- Reduction
- K-Means Clustering
- Histogram
- Linear Regression
- Logistic Regression

Previous manual UPMEM implementations of the same applications can be found in PrIM benchmark ([https://github.com/CMU-SAFARI/prim-benchmarks](https://github.com/CMU-SAFARI/prim-benchmarks)), dpu_kmeans ([https://github.com/upmem/dpu_kmeans](https://github.com/upmem/dpu_kmeans)) and prim-ml ([https://github.com/CMU-SAFARI/prim-ml](https://github.com/CMU-SAFARI/prim-ml)). These previous implementations can serve as baseline for measuring SimplePIM's performance as well as productivity improvements.
Conclusion

• **Processing-in-Memory (PIM)** promises to alleviate the *data movement bottleneck*

• Real PIM hardware is now available, e.g., UPMEM PIM

• However, **programming real PIM hardware is challenging**, e.g.:
  - Distribute data across PIM memory banks,
  - Manage data transfers between host cores and PIM cores, and between PIM cores,
  - Launch PIM kernels on the PIM cores, etc.

• **SimplePIM** is a high-level programming framework for real PIM hardware
  - *Iterators* such as `map`, `reduce`, and `zip`
  - *Collective communication* with `broadcast`, `scatter`, and `gather`

• Implementation on UPMEM and evaluation with six different workloads
  - Reduction, vector add, histogram, linear/logistic regression, K-means
  - 4.4x fewer lines of code compared to hand-optimized code
  - Between 15% and 43% faster than hand-optimized code for three workloads

• Source code: [https://github.com/CMU-SAFARI/SimplePIM](https://github.com/CMU-SAFARI/SimplePIM)
Adoption: How to Maintain Coherence? (I)

- Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu,

"LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory"


LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory

Amirali Boroumand†, Saugata Ghose†, Minesh Patel†, Hasan Hassan†§, Brandon Lucia†, Kevin Hsieh†, Krishna T. Malladi*, Hongzhong Zheng*, and Onur Mutlu††

†Carnegie Mellon University  *Samsung Semiconductor, Inc.  §TOBB ETÜ  ††ETH Zürich
Challenge: Coherence for Hybrid CPU-PIM Apps

- Traditional coherence
- No coherence overhead

<table>
<thead>
<tr>
<th>Components</th>
<th>Radii</th>
<th>PageRank</th>
<th>Components</th>
<th>Radii</th>
<th>PageRank</th>
<th>Components</th>
<th>Radii</th>
<th>PageRank</th>
<th>HTAP-256</th>
<th>HTAP-128</th>
<th>GMean</th>
</tr>
</thead>
<tbody>
<tr>
<td>arXiv</td>
<td></td>
<td></td>
<td>Gnutella</td>
<td></td>
<td></td>
<td>Enron</td>
<td></td>
<td></td>
<td>IMDB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- CPU-only
- FG
- CG
- NC
- LazyPIM
- Ideal-PIM
Adoption: How to Maintain Coherence? (II)

- Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu,

"CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators"


CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators

Amirali Boroumand†
Brandon Lucia†
Saugata Ghose†
Rachata Ausavarungnirun†‡
Nastaran Hajinazar°†
Krishna T. Malladi§
Hasan Hassan*
Kevin Hsieh†
Hongzhong Zheng§
Onur Mutlu*†

†Carnegie Mellon University
°Simon Fraser University
‡ETH Zürich
*KMUTNB
§Samsung Semiconductor, Inc.
Adoption: How to Support Synchronization?

  - Slides (pptx) (pdf)
  - Short Talk Slides (pptx) (pdf)
  - Talk Video (21 minutes)
  - Short Talk Video (7 minutes)

**SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures**

Christina Giannoula†‡, Nandita Vijaykumar*‡, Nikela Papadopoulou†, Vasileios Karakostas†, Ivan Fernandez§‡, Juan Gómez-Luna†, Lois Orosa†, Nectarios Koziris†, Georgios Goumas†, Onur Mutlu‡

†National Technical University of Athens  ‡ETH Zürich  *University of Toronto  §University of Malaga
Adoption: How to Support Virtual Memory?

Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu,
"Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation"
Proceedings of the 34th IEEE International Conference on Computer Design (ICCD), Phoenix, AZ, USA, October 2016.

Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation

Kevin Hsieh† Samira Khan‡ Nandita Vijaykumar†
Kevin K. Chang† Amirali Boroumand† Saugata Ghose† Onur Mutlu§†
†Carnegie Mellon University ‡University of Virginia §ETH Zürich
Adoption: Code and Data Mapping


[Slides (pptx) (pdf)]
[Lightning Session Slides (pptx) (pdf)]
DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

GERALDO F. OLIVEIRA, ETH Zürich, Switzerland
JUAN GÓMEZ-LUNA, ETH Zürich, Switzerland
LOIS OROSA, ETH Zürich, Switzerland
SAUGATA GHOSE, University of Illinois at Urbana–Champaign, USA
NANDITA VIJAYKUMAR, University of Toronto, Canada
IVAN FERNANDEZ, University of Malaga, Spain & ETH Zürich, Switzerland
MOHAMMAD SADROSADATI, Institute for Research in Fundamental Sciences (IPM), Iran & ETH Zürich, Switzerland
ONUR MUTLU, ETH Zürich, Switzerland

Data movement between the CPU and main memory is a first-order obstacle against improving performance, scalability, and energy efficiency in modern systems. Computer systems employ a range of techniques to reduce overheads tied to data movement, spanning from traditional mechanisms (e.g., deep multi-level cache hierarchies, aggressive hardware prefetchers) to emerging techniques such as Near-Data Processing (NDP), where some computation is moved close to memory. Prior NDP works investigate the root causes of data movement bottlenecks using different profiling methodologies and tools. However, there is still a lack of understanding about the key metrics that can identify different data movement bottlenecks and their relation to traditional and emerging data movement mitigation mechanisms. Our goal is to methodically identify potential sources of data movement over a broad set of applications and to comprehensively compare traditional compute-centric data movement mitigation techniques (e.g., caching and prefetching) to more memory-centric techniques (e.g., NDP), thereby developing a rigorous understanding of the best techniques to mitigate each source of data movement.

With this goal in mind, we perform the first large-scale characterization of a wide variety of applications, across a wide range of application domains, to identify fundamental program properties that lead to data movement to/from main memory. We develop the first systematic methodology to classify applications based on the sources contributing to data movement bottlenecks. From our large-scale characterization of 77K functions across 345 applications, we select 144 functions to form the first open-source benchmark suite (DAMOV) for main memory data movement studies. We select a diverse range of functions that (1) represent different types of data movement bottlenecks, and (2) come from a wide range of application domains. Using NDP as a case study, we identify new insights about the different data movement bottlenecks and use these insights to determine the most suitable data movement mitigation mechanism for a particular application. We open-source DAMOV and the complete source code for our new characterization methodology at https://github.com/CMU-SAFARI/DAMOV.
Identifying Memory Bottlenecks

- Multiple approaches to identify applications that:
  - suffer from data movement bottlenecks
  - take advantage of NDP

- Existing approaches are not comprehensive enough

Roofline model

Arithmetic Intensity (OPS/byte)  Performance (GOPS/s)

High LLC MPKI

NDP Speedup over CPU  Last-Level Cache MPKI

SAFARI
Limitations of Prior Approaches (1/2)

- **Roofline model** → identifies when an application is *bounded* by *compute* or *memory* units

  - **Compute Roof**
    - $y = \text{Peak System Throughput}$

  - **Memory Roof**
    - $y = \text{BW} \times \text{AI}$

  - **Compute Bound** → Not suitable for NDP
  - **Memory Bound** → Suitable for NDP
• **Roofline model** → identifies when an application is *bounded* by **compute** or **memory** units
Limitations of Prior Approaches (1/2)

- **Roofline model** → identifies when an application is bounded by **compute** or **memory** units.

- **Memory Bound** applications are **faster on CPU**, or performance **depends**.

- **Compute Bound** applications are **faster on CPU**.

  - Compute Bound applications have **similar performance** on CPU/NDP or performance **depends**.

  - Memory Bound applications are **faster on NDP**.

  - Memory Bound applications are **faster on CPU**, or performance **depends**.

**Diagram Notes:**
- The diagram illustrates the performance of applications on CPU and NDP based on their arithmetic intensity.
- The performance is measured in GOPS/second.
Limitations of Prior Approaches (1/2)

- **Roofline model** → identifies when an application is bounded by compute or memory units

  - **Memory Bound** applications are faster on CPU
  - **Compute Bound** applications are faster on CPU, or performance depends

  Roofline model **does not accurately account** for the **NDP suitability** of memory-bound applications
Limitations of Prior Approaches (2/2)

- Application with a last-level cache MPKI > 10 → memory intensive and benefits from NDP

![Diagram showing NDP Speedup over CPU vs LLC MPKI]

- Faster on CPU
- Faster on NDP
- Similar on CPU/NDP
- Depends
Limitations of Prior Approaches (2/2)

- Application with a last-level cache MPKI $> 10$ → memory intensive and benefits from NDP

Applications with low MPKI can be faster on NDP; have similar performance on CPU/NDP or; performance can depends

Applications with low MPKI are faster on CPU

Applications with high MPKI are faster on NDP

SAFARI
Limitations of Prior Approaches (2/2)

- Application with a last-level cache MPKI > 10 → memory intensive and benefits from NDP

Applications with low MPKI can be faster on NDP;

LLC MPKI does not accurately account for the NDP suitability of memory-bound applications
Identifying Memory Bottlenecks

- Multiple approaches to identify applications that:
  - suffer from data movement bottlenecks
  - take advantage of NDP

- Existing approaches are not comprehensive enough
The Problem

• Multiple approaches to identify applications that:
  - suffer from data movement bottlenecks
  - take advantage of NDP

No available methodology can comprehensively:

– **identify** data movement bottlenecks

– **correlate** them with the **most suitable** data movement mitigation mechanism
Our Goal

- **Our Goal:** develop a methodology to:
  - methodically identify sources of data movement bottlenecks
  - comprehensively compare compute- and memory-centric data movement mitigation techniques
Methodology Overview

**Step 1: Application Profiling**
- **Profiler**: Analyzes memory traces.
  - roi_begin
  - roi_end

**Memory Traces**
- Temp.
- Locality

**Step 2: Locality-based Clustering**
- DRAM Bandwidth
- DRAM Latency
- L1/L2 Cache Capacity
- L3 Cache Contention
- Compute-bound

**Step 3: Memory Bottleneck Class**

**DAMOV-SIM Simulator**
- ld 0xFF
- st 0xAF
- ld 0xFF
- st 0xAF
- ld 0xFF

**User Input**
- Target Application
- Source Code

**Methodology Output**
- Memory Bottleneck Classes
- SAFARI
Methodology Overview

User Input
- Target Application
- Source Code

Step 1
Application Profiling
- Profiler
  - roi_begin
  - roi_end

Step 2
Locality-based Clustering
- High
- Low

Step 3
Memory Bottleneck Class.

Methodology Output
- Memory Bottleneck Classes

DAMOV-SIM Simulator
- ld 0xFF
- st 0xAF
- ld 0xFF
- st 0xAF
- ld 0xFF

Memory Traces
- Scalability Analysis
- # Cores

Profiler

SAFARI
Step 1: Application Profiling

**Goal:** Identify **application functions** that suffer from **data movement bottlenecks**

Hardware Profiling Tool: Intel VTune

**MemoryBound:**
CPU is stalled due to load/store
Methodology Overview

**Step 1: Application Profiling**
- **Memory Traces**
  - ld 0xFF
  - st 0xAF
  - ld 0xFF
  - st 0xAF
  - ld 0xFF

**Profiler**
- roi_begin
- roi_end

**Step 2: Locality-based Clustering**
- Temporal Locality
- Spatial Locality

**Step 3: Memory Bottleneck Class**
- High
  - Temp. Locality
  - Low
- Low
Step 2: Locality-Based Clustering

- **Goal:** analyze application’s memory characteristics

**Spatial Locality**

![Memory Trace]

- stride profile(1) += 1

---

**High spatial locality**

**Low spatial locality**

Step 2: Locality-Based Clustering

**Goal:** analyze application’s memory characteristics

**Spatial Locality**

- **Stride Profile Histogram:**
  - Low spatial locality
  - High spatial locality

- **Reuse Profile Histogram:**
  - Low temporal locality
  - High temporal locality

**Temporal Locality**

- **Memory Trace:**
  - Stride profile(1) += 1
  - Reuse profile(4) += 1

Methodology Overview

User Input
- Target Application
- Source Code

Step 1
Application Profiling
Profiler
- roi_begin
- roi_end

Step 2
Locality-based Clustering
- High LFMR
- Low LFMR

Step 3
Memory Bottleneck Class.
- Arithmetic Intensity
- LLC MPKI
- Last-to-First Miss Ratio (LFMR)

DAMOV-SIM Simulator
- ld 0xFF
- st 0xAF
- ld 0xFF
- st 0xAF
- ld 0xFF

Memory Traces
- Scalability Analysis

Methodology Output
- Memory Bottleneck Classes

SAFARI
Step 3: Memory Bottleneck Classification (1/2)

**Arithmetic Intensity (AI)**
- floating-point/arithmetic operations per L1 cache lines accessed
  → shows computational intensity per memory request

**LLC Misses-per-Kilo-Instructions (MPKI)**
- LLC misses per one thousand instructions
  → shows memory intensity

**Last-to-First Miss Ratio (LFMR)**
- LLC misses per L1 misses
  → shows if an application benefits from L2/L3 caches
Step 3: Memory Bottleneck Classification (2/2)

- **Goal:** identify the specific sources of data movement bottlenecks

DAMOV-SIM Simulator

Integrated ZSim and Ramulator

Scalability Analysis

- **Scalability Analysis:**
  - 1, 4, 16, 64, and 256 out-of-order/in-order host and NDP CPU cores
  - 3D-stacked memory as main memory

DAMOV-SIM: [https://github.com/CMU-SAFARI/DAMOV](https://github.com/CMU-SAFARI/DAMOV)

Configuration 1: Host CPU System

Configuration 2: NDP System
Step 3: Memory Bottleneck Analysis

Six classes of data movement bottlenecks:

- each class ↔ data movement mitigation mechanism

Memory Bottleneck Class

1a: DRAM Bandwidth
1b: DRAM Latency
1c: L1/L2 Cache Capacity
2a: L3 Cache Contention
2b: L1 Cache Capacity
2c: Compute-Bound
DAMOV is Open Source

• We open-source our benchmark suite and our toolchain

DAMOV-SIM

DAMOV Benchmarks

DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

DAMOV is a benchmark suite and a methodical framework targeting the study of data movement bottlenecks in modern applications. It is intended to study new architectures, such as near-data processing. The DAMOV benchmark suite is the first open-source benchmark suite for main memory data movement-related studies, based on our systematic characterization methodology. This suite consists of 144 functions representing different sources of data movement bottlenecks and can be used as a baseline benchmark set for future data-movement mitigation research. The applications in the DAMOV benchmark suite belong to popular benchmark suites, including BWA, Chai, Darknet, GASE, Hardware Effects, Hashjoin, HPCC, HPCG, Ligra, PARSEC, Parboil, PolyBench, Phoenix, Rodinia, SPLASH-2, STREAM.
DAMOV is Open Source

• We open-source our benchmark suite and our toolchain

Get DAMOV at:
https://github.com/CMU-SAFARI/DAMOV

DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

DAMOV is a benchmark suite and a methodical framework targeting the study of data movement bottlenecks in modern applications. It is intended to study new architectures, such as near-data processing.

The DAMOV benchmark suite is the first open-source benchmark suite for main memory data movement-related studies, based on our systematic characterization methodology. This suite consists of 144 functions representing different sources of data movement bottlenecks and can be used as a baseline benchmark set for future data-movement mitigation research. The applications in the DAMOV benchmark suite belong to popular benchmark suites, including BWA, Chai, Darknet, GASE, Hardware Effects, Hashjoin, HPCC, HPCG, Ligra, PARSEC, Parboil, PolyBench, Phoenix, Rodinia, SPLASH-2, STREAM.
More on DAMOV Analysis Methodology & Workloads

**Step 3: Memory Bottleneck Classification (2/2)**

- **Goal:** identify the specific sources of data movement bottlenecks

**Scalability Analysis:**
- 1, 4, 16, 64, and 256 out-of-order/in-order host and NDP CPU cores
- 3D-stacked memory as main memory

SAFARI Live Seminar: DAMOV: A New Methodology & Benchmark Suite for Data Movement Bottlenecks

352 views • Streamed live on Jul 22, 2021

[https://www.youtube.com/watch?v=GWideVyo0nM&list=PL5Q2soXY2Zi_tOTAYm--dYByNPL7JhwR9&index=3](https://www.youtube.com/watch?v=GWideVyo0nM&list=PL5Q2soXY2Zi_tOTAYm--dYByNPL7JhwR9&index=3)


[arXiv preprint]
[IEEE Access version]
[DAMOV Suite and Simulator Source Code]
[SAFARI Live Seminar Video (2 hrs 40 mins)]
[Short Talk Video (21 minutes)]

DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks

GERALDO F. OLIVEIRA, ETH Zürich, Switzerland
JUAN GÓMEZ-LUNA, ETH Zürich, Switzerland
LOIS OROSA, ETH Zürich, Switzerland
SAUGATA GHOSE, University of Illinois at Urbana–Champaign, USA
NANDITA VIJAYKUMAR, University of Toronto, Canada
IVAN FERNANDEZ, University of Malaga, Spain & ETH Zürich, Switzerland
MOHAMMAD SADROSADATI, ETH Zürich, Switzerland
ONUR MUTLU, ETH Zürich, Switzerland
Tutorial on Memory-Centric Computing: PIM Adoption & Programmability

Geraldo F. Oliveira
Prof. Onur Mutlu

ISCA 2024
29 June 2024