Sub-Agenda: In-Memory Computation

- Major Trends Affecting Main Memory
- **The Need for Intelligent Memory Controllers**
  - Bottom Up: Push from Circuits and Devices
  - Top Down: Pull from Systems and Applications
- Processing in Memory: Two Directions
  - Minimally Changing Memory Chips
  - Exploiting 3D-Stacked Memory
- How to Enable Adoption of Processing in Memory
- Conclusion
Three Key Systems Trends

1. Data access is a major bottleneck
   - Applications are increasingly data hungry

2. Energy consumption is a key limiter

3. Data movement energy dominates compute
   - Especially true for off-chip to on-chip movement
Observation and Opportunity

- High latency and high energy caused by data movement
  - Long, energy-hungry interconnects
  - Energy-hungry electrical interfaces
  - Movement of large amounts of data

- Opportunity: Minimize data movement by performing computation directly (near) where the data resides
  - Processing in memory (PIM)
  - In-memory computation/processing
  - Near-data processing (NDP)
  - General concept applicable to any data storage & movement unit (caches, SSDs, main memory, network, controllers)
Four Key Issues in Future Platforms

- Fundamentally Secure/Reliable/Safe Architectures

- Fundamentally Energy-Efficient Architectures
  - Memory-centric (Data-centric) Architectures

- Fundamentally Low-Latency Architectures

- Architectures for Genomics, Medicine, Health
Maslow’s (Human) Hierarchy of Needs, Revisited


Everlasting energy

Source: https://www.simplypsychology.org/maslow.html
Do We Want This?

Source: V. Milutinovic
Challenge and Opportunity for Future

High Performance, Energy Efficient, Sustainable
The Problem

Data access is the major performance and energy bottleneck

Our current design principles cause great energy waste
(and great performance loss)
The Problem

Processing of data is performed far away from the data
A Computing System

- Three key components
- Computation
- Communication
- Storage/memory

Burks, Goldstein, von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” 1946.

A Computing System

- Three key components
  - Computation
  - Communication
  - Storage/memory

Burks, Goldstein, von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” 1946.

Today’s Computing Systems

- Are overwhelmingly processor centric
- All data processed in the processor → at great system cost
- Processor is heavily optimized and is considered the master
- Data storage units are dumb and are largely unoptimized (except for some that are on the processor die)
Yet …

- “It’s the Memory, Stupid!” (Richard Sites, MPR, 1996)

The Performance Perspective

- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors"

Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors

Onur Mutlu § Jared Stark † Chris Wilkerson ‡ Yale N. Patt §

§ECE Department
The University of Texas at Austin
{onur,patt}@ece.utexas.edu

†Microprocessor Research
Intel Labs
jared.w.stark@intel.com

‡Desktop Platforms Group
Intel Corporation
chris.wilkerson@intel.com
The Performance Perspective (Today)

- All of Google’s Data Center Workloads (2015):

The Performance Perspective (Today)

- All of Google’s Data Center Workloads (2015):

![Box plot showing cache-bound cycles]

**Figure 11: Half of cycles are spent stalled on caches.**

Perils of Processor-Centric Design

- **Grossly-imbalanced systems**
  - Processing done only in **one place**
  - Everything else just stores and moves data: **data moves a lot**
    - Energy inefficient
    - Low performance
    - Complex

- **Overly complex and bloated processor (and accelerators)**
  - To tolerate data access from memory
  - Complex hierarchies and mechanisms
    - Energy inefficient
    - Low performance
    - Complex
Most of the system is dedicated to storing and moving data
The Energy Perspective

Communication Dominates Arithmetic

Dally, HiPEAC 2015
Data Movement vs. Computation Energy

Communication Dominates Arithmetic

A memory access consumes $\sim 1000X$ the energy of a complex addition

Dally, HiPEAC 2015
Data Movement vs. Computation Energy

- **Data movement** is a major system energy bottleneck
  - Comprises 41% of mobile system energy during web browsing [2]
  - Costs ~115 times as much energy as an ADD operation [1, 2]

[1]: Reducing data Movement Energy via Online Data Clustering and Encoding (MICRO’16)
[2]: Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms (IISWC’14)
Energy Waste in Mobile Devices


62.7% of the total system energy is spent on data movement
We Do Not Want to Move Data!

A memory access consumes $\sim 1000X$ the energy of a complex addition
We Need A Paradigm Shift To …

- Enable computation with minimal data movement
- Compute where it makes sense (where data resides)
- Make computing architectures more data-centric
Goal: Processing Inside Memory

Many questions ... How do we design the:
- compute-capable memory & controllers?
- processor chip and in-memory units?
- software and hardware interfaces?
- system software and languages?
- algorithms?
Why In-Memory Computation Today?

- **Push from Technology**
  - DRAM scaling at jeopardy
  - Controllers close to DRAM
  - Industry open to new memory architectures

- **Pull from Systems and Applications**
  - Data access is a major system and application bottleneck
  - Systems are energy limited
  - Data movement much more energy-hungry than computation
Sub-Agenda: In-Memory Computation

- Major Trends Affecting Main Memory
- The Need for Intelligent Memory Controllers
  - Bottom Up: Push from Circuits and Devices
  - Top Down: Pull from Systems and Applications
- Processing in Memory: Two Directions
  - Minimally Changing Memory Chips
  - Exploiting 3D-Stacked Memory
- How to Enable Adoption of Processing in Memory
- Conclusion
Processing in Memory: Two Approaches

1. Minimally changing memory chips
2. Exploiting 3D-stacked memory
Approach 1: Minimally Changing DRAM

- DRAM has great capability to perform **bulk data movement and computation** internally with small changes
  - Can exploit internal connectivity to move data
  - Can exploit analog computation capability
  - ...

Examples: RowClone, In-DRAM AND/OR, Gather/Scatter DRAM

- **RowClone**: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data (Seshadri et al., MICRO 2013)
- **Fast Bulk Bitwise AND and OR in DRAM** (Seshadri et al., IEEE CAL 2015)
- **Gather-Scatter DRAM**: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses (Seshadri et al., MICRO 2015)
- "**Ambit**: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology" (Seshadri et al., MICRO 2017)
Starting Simple: Data Copy and Initialization

Bulk Data Copy

Bulk Data Initialization
Bulk Data Copy and Initialization

The Impact of Architectural Trends on Operating System Performance
Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta

Hardware Support for Bulk Data Movement in Server Platforms
Li Zhao†, Ravi Iyer‡, Srihari Makineni‡, Laxmi Bhuyan† and Don Newell‡
†Department of Computer Science and Engineering, University of California, Riverside, CA 92521
Email: {zhao, bhuyan}@cs.ucr.edu
‡Communications Technology Lab, Intel Corporation

Architecture Support for Improving Bulk Memory Copying and Initialization Performance
Xiaowei Jiang, Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University
Raleigh, USA

Li Zhao, Ravishankar Iyer
Intel Labs
Intel Corporation
Hillsboro, USA

SAFARI
Starting Simple: Data Copy and Initialization

*memmove* & *memcpy*: 5% cycles in Google’s datacenter [Kanev+ ISCA’15]

Forking

Zero initialization (e.g., security)

Checkpointing

VM Cloning

Deduplication

Page Migration

Many more
Today’s Systems: Bulk Data Copy

1) High latency
2) High bandwidth utilization
3) Cache pollution
4) Unwanted data movement

1046ns, 3.6uJ (for 4KB page copy via DMA)
Future Systems: In-Memory Copy

1) Low latency
2) Low bandwidth utilization
3) No cache pollution
4) No unwanted data movement

1046ns, 3.6uJ → 90ns, 0.04uJ
RowClone: In-DRAM Row Copy

Idea: Two consecutive ACTivates
Negligible HW cost

11.6X latency reduction, 74X energy reduction
RowClone: Intra-Subarray

Data gets copied

\[ V_{DD}/2 + \delta \]

Amplify the difference

Sense Amplifier (Row Buffer)

0
RowClone: Intra-Subarray (II)

1. **Activate** src row (copy data from src to row buffer)

2. **Activate** dst row (disconnect src from row buffer, connect dst – copy data from row buffer to dst)
RowClone: Inter-Bank

Overlap the latency of the read and the write
1.9X latency reduction, 3.2X energy reduction
Generalized RowClone

- Inter Subarray Copy (Use Inter-Bank Copy Twice)
- Inter Bank Copy (Pipelined Internal RD/WR)
- Intra Subarray Copy (2 ACTs)

0.01% area cost
RowClone: Fast Row Initialization

Fix a row at Zero
(0.5% loss in capacity)
RowClone: Bulk Initialization

- Initialization with arbitrary data
  - Initialize one row
  - Copy the data to other rows

- Zero initialization (most common)
  - Reserve a row in each subarray (always zero)
  - Copy data from reserved row (FPM mode)
  - 6.0x lower latency, 41.5x lower DRAM energy
  - 0.2% loss in capacity
RowClone: Latency & Energy Benefits

**Latency Reduction**
- Copy: 11.6x
- Zero: 6.0x

**Energy Reduction**
- Copy: 74.4x
- Zero: 41.5x

**Very low cost: 0.01% increase in die area**
Copy and Initialization in Workloads

![Fraction of Memory Traffic](image)

- **Zero**
- **Copy**
- **Write**
- **Read**

- **Bootup**
- **Compile**
- **Forkbench**
- **Mcached**
- **Mysql**
- **Shell**
RowClone: Application Performance

% Compared to Baseline

- IPC Improvement
- Energy Reduction

bootup | compile | forkbench | mcached | mysql | shell

energy reduction
End-to-End System Design

- Application
- Operating System
- ISA
- Microarchitecture
- DRAM (RowClone)

How to communicate occurrences of bulk copy/initialization across layers?

How to ensure cache coherence?

How to maximize latency and energy savings?

How to handle data reuse?
More on RowClone

- Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry,

"RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization"

Proceedings of the 46th International Symposium on Microarchitecture (MICRO), Davis, CA, December 2013. [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
Memory as an Accelerator

Memory similar to a “conventional” accelerator
In-Memory Bulk Bitwise Operations

- We can support in-DRAM COPY, ZERO, AND, OR, NOT, MAJ
- At low cost
- Using analog computation capability of DRAM
  - Idea: activating multiple rows performs computation
- 30-60X performance and energy improvement

- New memory technologies enable even more opportunities
  - Memristors, resistive RAM, phase change mem, STT-MRAM, ...
  - Can operate on data with minimal movement
In-DRAM AND/OR: Triple Row Activation

\[ \frac{1}{2}V_{DD} + \delta \]

**Final State**

\[ AB + BC + AC \]

\[ C(A + B) + \sim C(AB) \]

In-DRAM Bulk Bitwise AND/OR Operation

- **BULKAND A, B → C**

  Semantics: Perform a bitwise AND of two rows A and B and store the result in row C

- R0 – reserved zero row, R1 – reserved one row
- D1, D2, D3 – Designated rows for triple activation

1. RowClone A into D1
2. RowClone B into D2
3. RowClone R0 into D3
4. ACTIVATE D1, D2, D3
5. RowClone Result into C
More on In-DRAM Bulk AND/OR

- Vivek Seshadri, Kevin Hsieh, Amirali Boroumand, Donghyuk Lee, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry,

"Fast Bulk Bitwise AND and OR in DRAM"

In-DRAM NOT: Dual Contact Cell

Idea:
Feed the negated value in the sense amplifier into a special row

Figure 5: A dual-contact cell connected to both ends of a sense amplifier

In-DRAM NOT Operation

Performance: In-DRAM Bitwise Operations

Figure 9: Throughput of bitwise operations on various systems.

# Energy of In-DRAM Bitwise Operations

<table>
<thead>
<tr>
<th>Design</th>
<th>not</th>
<th>and/or</th>
<th>nand/nor</th>
<th>xor/xnor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DRAM &amp; Channel Energy</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDR3</td>
<td>93.7</td>
<td>137.9</td>
<td>137.9</td>
<td>137.9</td>
</tr>
<tr>
<td>Ambit</td>
<td>1.6</td>
<td>3.2</td>
<td>4.0</td>
<td>5.5</td>
</tr>
<tr>
<td>(nJ/KB)</td>
<td>59.5X</td>
<td>43.9X</td>
<td>35.1X</td>
<td>25.1X</td>
</tr>
</tbody>
</table>

Table 3: Energy of bitwise operations. (↓) indicates energy reduction of Ambit over the traditional DDR3-based design.

Ambit vs. DDR3: Performance and Energy

Bulk Bitwise Operations in Workloads

- Bitmap indices (database indexing)
- Set operations
- Encryption algorithms
- BitWeaving (database queries)
- BitFunnel (web search)
- DNA sequence mapping

[1] Li and Patel, BitWeaving, SIGMOD 2013
Example Data Structure: Bitmap Index

- Alternative to B-tree and its variants
- Efficient for performing *range queries* and *joins*
- Many bitwise operations to perform a query

- age < 18
- 18 < age < 25
- 25 < age < 60
- age > 60
Performance: Bitmap Index on Ambit

![Bar Chart]

**Figure 10:** Bitmap index performance. The value above each bar indicates the reduction in execution time due to Ambit.

Performance: BitWeaving on Ambit

Figure 11: Speedup offered by Ambit over baseline CPU with SIMD for BitWeaving

More on In-DRAM Bulk AND/OR

- Vivek Seshadri, Kevin Hsieh, Amirali Boroumand, Donghyuk Lee, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry,

"Fast Bulk Bitwise AND and OR in DRAM"


---

Fast Bulk Bitwise AND and OR in DRAM

Vivek Seshadri*, Kevin Hsieh*, Amirali Boroumand*, Donghyuk Lee*, Michael A. Kozuch†, Onur Mutlu*, Phillip B. Gibbons†, Todd C. Mowry*

* Carnegie Mellon University
† Intel Pittsburgh
More on In-DRAM Bitwise Operations


Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology

Vivek Seshadri\textsuperscript{1,5} Donghyuk Lee\textsuperscript{2,5} Thomas Mullins\textsuperscript{3,5} Hasan Hassan\textsuperscript{4} Amirali Boroumand\textsuperscript{5}
Jeremie Kim\textsuperscript{4,5} Michael A. Kozuch\textsuperscript{3} Onur Mutlu\textsuperscript{4,5} Phillip B. Gibbons\textsuperscript{5} Todd C. Mowry\textsuperscript{5}

\textsuperscript{1}Microsoft Research India \textsuperscript{2}NVIDIA Research \textsuperscript{3}Intel \textsuperscript{4}ETH Zürich \textsuperscript{5}Carnegie Mellon University
Challenge: Intelligent Memory Device

Does memory have to be dumb?
Challenge and Opportunity for Future Computing Architectures with Minimal Data Movement
Agenda

- Major Trends Affecting Main Memory
- The Need for Intelligent Memory Controllers
  - Bottom Up: Push from Circuits and Devices
  - Top Down: Pull from Systems and Applications
- Processing in Memory: Two Directions
  - Minimally Changing Memory Chips
  - Exploiting 3D-Stacked Memory
- How to Enable Adoption of Processing in Memory
- Conclusion
Processing in Memory: Two Approaches

1. Minimally changing memory chips
2. Exploiting 3D-stacked memory
Opportunity: 3D-Stacked Logic+Memory

Other “True 3D” technologies under development
# DRAM Landscape (circa 2015)

<table>
<thead>
<tr>
<th>Segment</th>
<th>DRAM Standards &amp; Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commodity</td>
<td>DDR3 (2007) [14]; DDR4 (2012) [18]</td>
</tr>
<tr>
<td>Performance</td>
<td>eDRAM [28], [32]; RLDDRAM3 (2011) [29]</td>
</tr>
</tbody>
</table>

Table 1. Landscape of DRAM-based memory

Several Questions in 3D-Stacked PIM

- What are the performance and energy benefits of using 3D-stacked memory as a coarse-grained accelerator?
  - By changing the entire system
  - By performing simple function offloading

- What is the minimal processing-in-memory support we can provide?
  - With minimal changes to system and programming
Graph Processing

- Large graphs are everywhere (circa 2015)
  - 36 Million Wikipedia Pages
  - 1.4 Billion Facebook Users
  - 300 Million Twitter Users
  - 30 Billion Instagram Photos

- Scalable large-scale graph processing is challenging

<table>
<thead>
<tr>
<th>32 Cores</th>
<th>128...</th>
<th>+42%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  Speedup
Key Bottlenecks in Graph Processing

for (v: graph.vertices) {
  for (w: v.successors) {
    w.next_rank += weight * v.rank;
  }
}

1. Frequent random memory accesses
2. Little amount of computation
Tesseract System for Graph Processing

Interconnected set of 3D-stacked memory+logic chips with simple cores

Host Processor

Memory-Mapped Accelerator Interface
Noncacheable, Physically Addressed)

In-Order Core

Crossbar Network

Logic

Message Queue

DRAM Controller

SAFARI Ahn+, “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing” ISCA 2015.
Tesseract System for Graph Processing

Host Processor
Memory-Mapped Accelerator Interface
Noncacheable, Physically Addressed)

Memory
Logic

Crossbar Network

In-Order Core

Communications via Remote Function Calls

Message Queue
for (v: graph.vertices) {
    for (w: v.successors) {
        w.next_rank += weight * v.rank;
    }
}
for (v: graph.vertices) {
    for (w: v.successors) {
        w.next_rank += weight * v.rank;
    }
}

Vault #1

&v

Vault #2

w
for (v: graph.vertices) {
    for (w: v.successors) {
        put(w.id, function() { w.next_rank += weight * v.rank; });
    }
}
Remote Function Call (Non-Blocking)

1. Send function address & args to the remote core
2. Store the incoming message to the message queue
3. Flush the message queue when it is full or a synchronization barrier is reached

```
put(w.id, function() { w.next_rank += value; })
```
Tesseract System for Graph Processing

Host Processor

Memory-Mapped Accelerator Interface
(Noncacheable, Physically Addressed)

Memory

Crossbar Network

Logic

Prefetching

PF Buffer

MTP

Message Queue

DRAM Controller

NI
Evaluated Systems

<table>
<thead>
<tr>
<th>DDR3-OoO</th>
<th>HMC-OoO</th>
<th>HMC-MC</th>
<th>Tesseract</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram of DDR3-OoO" /></td>
<td><img src="image2" alt="Diagram of HMC-OoO" /></td>
<td><img src="image3" alt="Diagram of HMC-MC" /></td>
<td><img src="image4" alt="Diagram of Tesseract" /></td>
</tr>
<tr>
<td>8 OoO 4GHz</td>
<td>8 OoO 4GHz</td>
<td>128 In-Order 2GHz</td>
<td>32 Tesseract Cores</td>
</tr>
<tr>
<td>8 OoO 4GHz</td>
<td>8 OoO 4GHz</td>
<td>128 In-Order 2GHz</td>
<td></td>
</tr>
<tr>
<td>8 OoO 4GHz</td>
<td>8 OoO 4GHz</td>
<td>128 In-Order 2GHz</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>102.4GB/s</td>
<td>640GB/s</td>
<td>640GB/s</td>
<td>8TB/s</td>
</tr>
</tbody>
</table>

SAFARI Ahn+, “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing” ISCA 2015.
Tesseract Graph Processing Performance

>13X Performance Improvement

On five graph processing algorithms

- DDR3-OoO
- HMC-OoO
- HMC-MC
- Tesseract
- Tesseract-LP
- Tesseract-LP-MTP

>13X Performance Improvement

Ahn+, “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing” ISCA 2015.
Tesseract Graph Processing Performance

Memory Bandwidth Consumption

- DDR3-OoO: 80GB/s
- HMC-OoO: 190GB/s
- HMC-MC: 243GB/s
- Tesseract: 1.3TB/s
- Tesseract-LP: 2.2TB/s
- Tesseract-LP-MTP: 2.9TB/s
Effect of Bandwidth & Programming Model

- HMC-MC Bandwidth (640GB/s)
- Tesseract Bandwidth (8TB/s)

Bandwidth

Programming Model

HMC-MC
HMC-MC + PIM BW
Tesseract + Conventional BW
Tesseract (No Prefetching)

Speedup

2.3x
3.0x
6.5x
Tesseract Graph Processing System Energy

- Memory Layers
- Logic Layers
- Cores

> 8X Energy Reduction

Ahn+, “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing” ISCA 2015.
Tesseract: Advantages & Disadvantages

- **Advantages**
  + Specialized graph processing accelerator using PIM
  + Large system performance and energy benefits
  + Takes advantage of 3D stacking for an important workload
  + More general than just graph processing

- **Disadvantages**
  - Changes a lot in the system
    - New programming model
    - Specialized Tesseract cores for graph processing
  - Cost
  - Scalability limited by off-chip links or graph partitioning
More on Tesseract

- Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi,
  "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing"
  [Slides (pdf)] [Lightning Session Slides (pdf)]

A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing

Junwhan Ahn  Sungpack Hong§  Sungjoo Yoo  Onur Mutlu†  Kiyoung Choi
junwhan@snu.ac.kr, sungpack.hong@oracle.com, sungjoo.yoo@gmail.com, onur@cmu.edu, kchoi@snu.ac.kr
Seoul National University  §Oracle Labs  †Carnegie Mellon University
Several Questions in 3D-Stacked PIM

- What are the performance and energy benefits of using 3D-stacked memory as a coarse-grained accelerator?
  - By changing the entire system
  - By performing simple function offloading

- What is the minimal processing-in-memory support we can provide?
  - With minimal changes to system and programming
3D-Stacked PIM on Mobile Devices

- Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu,

"Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks"


Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks

Amirali Boroumand¹
Rachata Ausavarungnirun¹
Aki Kuusela³

Saugata Ghose¹
Eric Shiu³

Youngsok Kim²
Rahul Thakur³

Daehyun Kim⁴,³

Parthasarathy Ranganathan³

Onur Mutlu⁵,¹
Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks

Amirali Boroumand

Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, Onur Mutlu
Consumer Devices

Consumer devices are everywhere!

Energy consumption is a first-class concern in consumer devices.
Popular Google Consumer Workloads

Chrome
Google’s web browser

TensorFlow Mobile
Google’s machine learning framework

VP9
Video Playback
Google’s video codec

VP9
Video Capture
Google’s video codec
Energy Cost of Data Movement

1st key observation: 62.7% of the total system energy is spent on data movement

Potential solution: move computation close to data

Challenge: limited area and energy budget
Using PIM to Reduce Data Movement

2nd key observation: a significant fraction of the data movement often comes from simple functions.

We can design lightweight logic to implement these simple functions in memory.

Offloading to PIM logic reduces energy and improves performance, on average, by 55.4% and 54.2%.
Workload Analysis

Chrome
Google’s web browser

TensorFlow
Google’s machine learning framework

VP9
YouTube Video Playback
Google’s video codec

Video Capture
Google’s video Codec
Workload Analysis

Chrome
Google’s web browser

TensorFlow
Google’s machine learning framework

VP9
YouTube
Video Playback
Google’s video codec

VP9
YouTube
Video Capture
Google’s video codec
How Chrome Renders a Web Page

1. Loading and Parsing
2. HTML Parser
3. HTML
4. CSS Parser
5. CSS
6. Render Tree
7. Layout
8. Rasterization
9. Compositing

Diagram:
- HTML
- HTML Parser
- Render Tree
- Layout
- Rasterization
- Compositing

- CSS
- CSS Parser
How Chrome Renders a Web Page

Loading and Parsing

- HTML
  - HTML Parser
- CSS
  - CSS Parser

Layouting

- Render Tree
- Layout

Rasterization

- Paints those objects and generates the bitmaps

Compositing

- Assembles all layers into a final screen image

SAFARI
Browser Analysis

• To satisfy user experience, the browser must provide:
  – Fast loading of webpages
  – Smooth scrolling of webpages
  – Quick switching between browser tabs

• We focus on two important user interactions:
  1) Page Scrolling
  2) Tab Switching
  – Both include page loading
Tab Switching
What Happens During Tab Switching?

- **Chrome employs a multi-process architecture**
  - Each tab is a separate process

![Chrome Process Diagram]

- **Main operations during tab switching:**
  - Context switch
  - Load the new page
Memory Consumption

- Primary concerns during tab switching:
  - How fast a new tab *loads* and *becomes interactive*
  - Memory consumption

Chrome uses *compression* to reduce each tab’s *memory footprint*
Data Movement Study

• To study data movement during tab switching, we emulate a user switching through 50 tabs

We make two key observations:

1. Compression and decompression contribute to 18.1% of the total system energy

2. 19.6 GB of data moves between CPU and ZRAM
Can We Use PIM to Mitigate the Cost?

CPU-Only

Swap out N pages
Read N Pages
Compress
Write back
Other tasks

Uncompressed Pages

high data movement

CPU + PIM

Swap out N pages
Other tasks
Compress
ZRAM

Uncompressed Pages

No off-chip data movement

PIM core and PIM accelerator are feasible to implement in-memory compression/decompression
Tab Switching Wrap Up

A large amount of **data movement** happens during **tab switching** as Chrome attempts to **compress** and **decompress** tabs.

Both functions can benefit from PIM execution and can be implemented as PIM logic.
Workload Analysis

Chrome
Google’s web browser

TensorFlow Mobile
Google’s machine learning framework

VP9
Video Playback
Google’s video codec

VP9
Video Capture
Google’s video codec
57.3% of the inference energy is spent on data movement.

54.4% of the data movement energy comes from packing/unpacking and quantization.
Packing

Reorders elements of matrices to minimize cache misses during matrix multiplication

Up to 40% of the inference energy and 31% of inference execution time

Packing’s data movement accounts for up to 35.3% of the inference energy

A simple data reorganization process that requires simple arithmetic
Quantization

Converts **32-bit floating point** to **8-bit integers** to improve inference execution time and energy consumption.

- Up to 16.8% of the inference energy and 16.1% of inference execution time.
- Majority of quantization energy comes from data movement.

A simple data conversion operation that requires shift, addition, and multiplication operations.
Quantization

Converts 32-bit floating point to 8-bit integers to improve inference execution time and energy consumption.

Based on our analysis, we conclude that:

- Both functions are good candidates for PIM execution
- It is feasible to implement them in PIM logic

A simple data conversion operation that requires shift, addition, and multiplication operations.
Evaluation Methodology

- **System Configuration (gem5 Simulator)**
  - **SoC**: 4 OoO cores, 8-wide issue, 64 kB L1 cache, 2MB L2 cache
  - **PIM Core**: 1 core per vault, 1-wide issue, 4-wide SIMD, 32kB L1 cache
  - **3D-Stacked Memory**: 2GB cube, 16 vaults per cube
    - Internal Bandwidth: 256GB/S
    - Off-Chip Channel Bandwidth: 32 GB/s
  - **Baseline Memory**: LPDDR3, 2GB, FR-FCFS scheduler

- **We study each target in isolation** and emulate each separately and run them in our simulator
Normalized Energy

**Normalized Energy**

```
<table>
<thead>
<tr>
<th>Process</th>
<th>CPU-Only</th>
<th>PIM-Core</th>
<th>PIM-Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texture</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Color</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compression</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decompression</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Packing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quantization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub-Pixel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deblocking</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motion</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

77.7% and 82.6% of energy reduction for **texture tiling** and **packing** comes from eliminating **data movement**

PIM core and PIM accelerator reduces **energy consumption** on average by **49.1%** and **55.4%**
Offloading these kernels to **PIM core** and **PIM accelerator** improves **performance** on average by **44.6%** and **54.2%**
Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks

Amirali Boroumand

Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, Onur Mutlu

ASPLOS 2018

SAFARI
Carnegie Mellon
Google
Samsung
ETH Zürich
62.7% of the total system energy is spent on data movement
Truly Distributed GPU Processing with PIM?

3D-stacked memory
(memory stack)

SM (Streaming Multiprocessor)

Main GPU

Logic layer

Crossbar switch

Vault Ctrl

Vault Ctrl

Logic layer

SM

global
void applyScaleFactorsKernel( uint8_T * const out,
uint8_T const * const in, const double *factor,
size_t const numRows, size_t const numCols )
{
    // Work out which pixel we are working on.
    const int rowIdx = blockIdx.x * blockDim.x + threadIdx.x;
    const int colIdx = blockIdx.y;
    const int sliceIdx = threadIdx.z;

    // Check this thread isn't off the image
    if( rowIdx >= numRows ) return;

    // Compute the index of my element
    size_t linearIdx = rowIdx + colIdx*numRows + sliceIdx*numRows*numCols;
Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler,

"Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems"


[Slides (pptx) (pdf)]
[Lightning Session Slides (pptx) (pdf)]
Accelerating GPU Execution with PIM (II)

  Proceedings of the 25th International Conference on Parallel Architectures and Compilation Techniques (PACT), Haifa, Israel, September 2016.

Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities

Ashutosh Pattnaik\textsuperscript{1} Xulong Tang\textsuperscript{1} Adwait Jog\textsuperscript{2} Onur Kayiran\textsuperscript{3}
Asit K. Mishra\textsuperscript{4} Mahmut T. Kandemir\textsuperscript{1} Onur Mutlu\textsuperscript{5,6} Chita R. Das\textsuperscript{1}

\textsuperscript{1}Pennsylvania State University \textsuperscript{2}College of William and Mary
\textsuperscript{3}Advanced Micro Devices, Inc. \textsuperscript{4}Intel Labs \textsuperscript{5}ETH Zürich \textsuperscript{6}Carnegie Mellon University

SAFARI
Accelerating Linked Data Structures

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
Executive Summary

• Our Goal: Accelerating pointer chasing inside main memory

• Challenges: Parallelism challenge and Address translation challenge

• Our Solution: In-Memory Pointer Chasing Accelerator (IMPICA)
  • Address-access decoupling: enabling parallelism in the accelerator with low cost
  • IMPICA page table: low cost page table in logic layer

• Key Results:
  • 1.2X – 1.9X speedup for pointer chasing operations, +16% database throughput
  • 6% - 41% reduction in energy consumption
Linked Data Structures

• Linked data structures are widely used in many important applications

Linked data structures are connected by pointers

B-Tree

Hash Table
The Problem: Pointer Chasing

• Traversing linked data structures requires chasing pointers

Find(A)

Serialized and irregular access pattern
6X cycles per instruction in real workloads
Our Goal

Accelerating pointer chasing inside main memory

Find(A)
Parallelism Challenge

- CPU core: Comp → Memory access → Comp
- CPU core: Comp → Memory access → Comp
- In-Memory Accelerator: Comp → Memory access → Comp
- In-Memory Accelerator: Comp → Memory access → Comp

**Slower for two operations**
Parallelism Challenge and Opportunity

• A simple in-memory accelerator can still be slower than multiple CPU cores

• **Opportunity**: a pointer-chasing accelerator spends a long time waiting for memory
Our Solution: Address-Access Decoupling

Address-access decoupling enables parallelism in both engines with low cost.
IMPICA Core Architecture

DRAM

DRAM Layers

Logic Layer

IMPICA Cache

Access Queue

Response Queue

Request Queue

Address Engine

Access Engine

Memory Controller

To / From CPU

128
Address Translation Challenge

The page table walk requires multiple memory accesses.

No TLB/MMU on the memory side.
Duplicating it is costly and creates compatibility issue.

Page table walk.
Our Solution: IMPICA Page Table

- Completely **decouple the page table of IMPICA from the page table of the CPUs**

<table>
<thead>
<tr>
<th>IMPICA Page Table</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Page</td>
<td>Physical Page</td>
</tr>
</tbody>
</table>

**Map linked data structure into IMPICA regions**

**IMPICA page table is a partial-to-any mapping**
Virtual Address

**Flat page table saves one memory access**

Tiny region table is almost always in the cache

Region Table

Flat Page Table (2MB)

Small Page Table (4KB)

Physical Address
Evaluation Methodology

• Simulator: gem5

• System Configuration
  • CPU
    • 4 OoO cores, 2GHz
    • Cache: 32KB L1, 1MB L2
  • IMPICA
    • 1 core, 500MHz, 32KB Cache

• Memory Bandwidth
  • 12.8 GB/s for CPU, 51.2 GB/s for IMPICA

• Our simulator code is open source
  • https://github.com/CMU-SAFARI/IMPICA
Result – Microbenchmark Performance

- Baseline + extra 128KB L2
- IMPICA

Speedup

Linked List: 1.9X
Hash Table: 1.3X
B-Tree: 1.2X
Result – Database Performance

**Database Throughput**
- Baseline + extra 128KB L2: +2%
- Baseline + extra 1MB L2: +5%
- IMPICA: +16%

**Database Latency**
- Baseline + extra 128KB L2: -0%
- Baseline + extra 1MB L2: -4%
- IMPICA: -13%
System Energy Consumption

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Baseline + extra 128KB L2</th>
<th>IMPICA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>-41%</td>
<td></td>
</tr>
<tr>
<td>Hash Table</td>
<td>-24%</td>
<td></td>
</tr>
<tr>
<td>B-Tree</td>
<td>-10%</td>
<td></td>
</tr>
<tr>
<td>DBx1000</td>
<td>-6%</td>
<td></td>
</tr>
</tbody>
</table>
## Area and Power Overhead

<table>
<thead>
<tr>
<th>Component</th>
<th>Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (Cortex-A57)</td>
<td>5.85 mm² per core</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>5 mm² per MB</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>10 mm²</td>
</tr>
<tr>
<td>IMPICA (+32KB cache)</td>
<td>0.45 mm²</td>
</tr>
</tbody>
</table>

- **Power overhead**: average power increases by 5.6%
Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller"
[Slides (pptx) (pdf)]
[Lightning Session Slides (pptx) (pdf)]
Automatic Offloading of Prefetch Mechanisms

- Milad Hashemi, Onur Mutlu, and Yale N. Patt,
  "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads"
  Proceedings of the 49th International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, October 2016.
  [Slides (pptx) (pdf)] [Lightning Session Slides (pdf)] [Poster (pptx) (pdf)]

Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads

  Milad Hashemi*, Onur Mutlu§, Yale N. Patt*

  *The University of Texas at Austin  §ETH Zürich
Several Questions in 3D-Stacked PIM

- What are the performance and energy benefits of using 3D-stacked memory as a coarse-grained accelerator?
  - By changing the entire system
  - By performing simple function offloading

- What is the **minimal processing-in-memory support** we can provide?
  - With minimal changes to system and programming
Memory Systems
and Memory-Centric Computing Systems

Lecture 3b: Processing-in-Memory I

Prof. Onur Mutlu
omutlu@gmail.com
https://people.inf.ethz.ch/omutlu
14 June 2019
TU Wien Fast Course 2019
Backup Slides
PIM-Enabled Instructions

- Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoungh Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture"
  [Slides (pdf)] [Lightning Session Slides (pdf)]
PEI: PIM-Enabled Instructions (Ideas)

- **Goal:** Develop mechanisms to get the most out of near-data processing with minimal cost, minimal changes to the system, no changes to the programming model

- **Key Idea 1:** Expose each PIM operation as a cache-coherent, virtually-addressed host processor instruction (called PEI) that operates on only a single cache block
  - e.g., __pim_add(&w.next_rank, value) \to pim.add r1, (r2)
  - No changes sequential execution/programming model
  - No changes to virtual memory
  - Minimal changes to cache coherence
  - No need for data mapping: Each PEI restricted to a single memory module

- **Key Idea 2:** Dynamically decide where to execute a PEI (i.e., the host processor or PIM accelerator) based on simple locality characteristics and simple hardware predictors
  - Execute each operation at the location that provides the best performance
for (v: graph.vertices) {
    value = weight * v.rank;
    for (w: v.successors) {
        w.next_rank += value;
    }
}

64 bytes in
64 bytes out
Simple PIM Operations as ISA Extensions (III)

```java
for (v: graph.vertices) {
    value = weight * v.rank;
    for (w: v.successors) {
        __pim_add(&w.next_rank, value);
    }
}
```

Host Processor

Main Memory

8 bytes in
0 bytes out

In-Memory Addition
Always Executing in Memory? Not A Good Idea

Increased Memory Bandwidth Consumption
Caching very effective

Reduced Memory Bandwidth Consumption due to
In-Memory Computation

More Vertices
PEI: PIM-Enabled Instructions (Example)

```c
for (v: graph.vertices) {
    value = weight * v.rank;
    for (w: v.successors) {
        __pim_add(&w.next_rank, value);
    }
}
pfence();
```

- Executed either in memory or in the processor: dynamic decision
  - Low-cost locality monitoring for a single instruction
- Cache-coherent, virtually-addressed, single cache block only
- Atomic between different PEIs
- *Not* atomic with normal instructions (use `pfence` for ordering)

<table>
<thead>
<tr>
<th>Operation</th>
<th>R</th>
<th>W</th>
<th>Input</th>
<th>Output</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-byte integer increment</td>
<td>O</td>
<td>O</td>
<td>0 bytes</td>
<td>0 bytes</td>
<td>AT</td>
</tr>
<tr>
<td>8-byte integer min</td>
<td>O</td>
<td>O</td>
<td>8 bytes</td>
<td>0 bytes</td>
<td>BFS, SP, WCC</td>
</tr>
<tr>
<td>Floating-point add</td>
<td>O</td>
<td>O</td>
<td>8 bytes</td>
<td>0 bytes</td>
<td>PR</td>
</tr>
<tr>
<td>Hash table probing</td>
<td>O</td>
<td>X</td>
<td>8 bytes</td>
<td>9 bytes</td>
<td>HJ</td>
</tr>
<tr>
<td>Histogram bin index</td>
<td>O</td>
<td>X</td>
<td>1 byte</td>
<td>16 bytes</td>
<td>HG, RP</td>
</tr>
<tr>
<td>Euclidean distance</td>
<td>O</td>
<td>X</td>
<td>64 bytes</td>
<td>4 bytes</td>
<td>SC</td>
</tr>
<tr>
<td>Dot product</td>
<td>O</td>
<td>X</td>
<td>32 bytes</td>
<td>8 bytes</td>
<td>SVM</td>
</tr>
</tbody>
</table>
PIM-Enabled Instructions

- Key to practicality: single-cache-block restriction
  - Each PEI can access *at most one last-level cache block*
  - Similar restrictions exist in atomic instructions

- Benefits
  - Localization: each PEI is bounded to one memory module
  - Interoperability: easier support for cache coherence and virtual memory
  - Simplified locality monitoring: data locality of PEIs can be identified simply by the cache control logic
Example (Abstract) PEI uArchitecture

Host Processor
- Out-Of-Order Core
- PCU (PEI Computation Unit)
- L1 Cache
- L2 Cache
- Last-Level Cache
- PMU (PEI Mgmt Unit)
- PIM Directory
- Locality Monitor
- HMC Controller

3D-stacked Memory
- PCU
- DRAM Controller
- Network
- 

Example PEI uArchitecture
PEI: Initial Evaluation Results

- Initial evaluations with **10 emerging data-intensive workloads**
  - Large-scale graph processing
  - In-memory data analytics
  - Machine learning and data mining
  - Three input sets (small, medium, large) for each workload to analyze the impact of data locality

- Pin-based cycle-level x86-64 simulation

**Performance Improvement and Energy Reduction:**
- 47% average speedup with large input data sets
- 32% speedup with small input data sets
- 25% avg. energy reduction in a single node with large input data sets

Table 2: Baseline Simulation Configuration

<table>
<thead>
<tr>
<th>Component</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>16 out-of-order cores, 4 GHz, 4-issue</td>
</tr>
<tr>
<td>L1 I/D-Cache</td>
<td>Private, 32 KB, 4/8-way, 64 B blocks, 16 MSHRs</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Private, 256 KB, 8-way, 64 B blocks, 16 MSHRs</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>Shared, 16 MB, 16-way, 64 B blocks, 64 MSHRs</td>
</tr>
<tr>
<td>On-Chip Network</td>
<td>Crossbar, 2 GHz, 144-bit links</td>
</tr>
<tr>
<td>Main Memory</td>
<td>32 GB, 8 HMCs, daisy-chain (80 GB/s full-duplex)</td>
</tr>
<tr>
<td>HMC</td>
<td>4 GB, 16 vaults, 256 DRAM banks [20]</td>
</tr>
<tr>
<td>– DRAM</td>
<td>FR-FCFS, tCL = tRCD = tRP = 13.75 ns [27]</td>
</tr>
<tr>
<td>– Vertical Links</td>
<td>64 TSVs per vault with 2 Gb/s signaling rate [23]</td>
</tr>
</tbody>
</table>
Evaluated Data-Intensive Applications

- Ten emerging data-intensive workloads
  - Large-scale graph processing
    - Average teenage follower, BFS, PageRank, single-source shortest path, weakly connected components
  - In-memory data analytics
    - Hash join, histogram, radix partitioning
  - Machine learning and data mining
    - Streamcluster, SVM-RFE

- Three input sets (small, medium, large) for each workload to show the impact of data locality
PEI Performance Delta: Large Data Sets

(Large Inputs, Baseline: Host-Only)
PEI Performance: Large Data Sets

![Normalized Amount of Off-chip Transfer](chart.png)
PEI Performance Delta: Small Data Sets

(Small Inputs, Baseline: Host-Only)

ATF  BFS  PR  SP  WCC  HJ  HG  RP  SC  SVM  GM

PIM-Only  Locality-Aware
PEI Performance: Small Data Sets

Normalized Amount of Off-chip Transfer

- ATF
- BFS
- PR
- SP
- WCC
- HJ
- HG
- RP
- SC
- SVM

Host-Only  PIM-Only  Locality-Aware
PEI Energy Consumption

Small
Medium
Large

- Host-Only
- PIM-Only
- Locality-Aware

- Cache
- HMC Link
- Host-side PCU
- Memory-side PCU
- DRAM
- PMU
PEI: Advantages & Disadvantages

- **Advantages**
  - Simple and low cost approach to PIM
  - No changes to programming model, virtual memory
  - Dynamically decides where to execute an instruction

- **Disadvantages**
  - Does not take full advantage of PIM potential
  - Single cache block restriction is limiting
Simpler PIM: PIM-Enabled Instructions

Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and KIyoung Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture"
[Slides (pdf)] [Lightning Session Slides (pdf)]
Automatic Code and Data Mapping

[Slides (pptx) (pdf)]
[Lightning Session Slides (pptx) (pdf)]
Automatic Offloading of Critical Code

- Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller". 
  [Slides (pptx) (pdf)] 
  [Lightning Session Slides (pptx) (pdf)]
Automatic Offloading of Prefetch Mechanisms

- Milad Hashemi, Onur Mutlu, and Yale N. Patt, "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads" in Proceedings of the 49th International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, October 2016. [Slides (pptx) (pdf)] [Lightning Session Slides (pdf)] [Poster (pptx) (pdf)]

Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads

Milad Hashemi*, Onur Mutlu§, Yale N. Patt*

*The University of Texas at Austin    §ETH Zürich
Efficient Automatic Data Coherence Support

- 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
Efficient Automatic Data Coherence Support

- 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\(^\d\) Brandon Lucia\(^\d\)
Saugata Ghose\(^\d\)
Rachata Ausavarungnirun\(^{\d\d}\)
Minesh Patel\(^*\)
Hasan Hassan\(^*\)
Nastaran Hajinazar\(^{\d\circ\d}\)
Krishna T. Malladi\(^$\)
Kevin Hsieh\(^{\d\d}\)
Hongzhong Zheng\(^$\)
Onur Mutlu\(^*^{\d\d}\)

\(\d\)Carnegie Mellon University
\(\circ\)Simon Fraser University
\(^{*}\)ETH Zürich
\(^{\d\d}\)KMUTNB
\(^$\)Samsung Semiconductor, Inc.
Challenge and Opportunity for Future

Fundamentally Energy-Efficient (Data-Centric) Computing Architectures
Challenge and Opportunity for Future

Fundamentally

High-Performance

(Data-Centric)

Computing Architectures
Challenge and Opportunity for Future Computing Architectures with Minimal Data Movement
Agenda

- Major Trends Affecting Main Memory
- The Need for Intelligent Memory Controllers
  - Bottom Up: Push from Circuits and Devices
  - Top Down: Pull from Systems and Applications
- Processing in Memory: Two Directions
  - Minimally Changing Memory Chips
  - Exploiting 3D-Stacked Memory
- How to Enable Adoption of Processing in Memory
- Conclusion
Eliminating the Adoption Barriers

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

1. Functionality of and applications for PIM

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

3. System support: coherence & virtual memory

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

5. Infrastructures to assess benefits and feasibility
We Need to Revisit the Entire Stack

We can get there step by step
Key Challenge 1: Code Mapping

- **Challenge 1:** Which operations should be executed in memory vs. in CPU?
Key Challenge 2: Data Mapping

- **Challenge 2**: How should data be mapped to different 3D memory stacks?

[Slides (pptx) (pdf)]
[Lightning Session Slides (pptx) (pdf)]
How to Schedule Code?


Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities

Ashutosh Pattnaik\(^1\)  Xulong Tang\(^1\)  Adwait Jog\(^2\)  Onur Kayiran\(^3\)  
Asit K. Mishra\(^4\)  Mahmut T. Kandemir\(^1\)  Onur Mutlu\(^5,6\)  Chita R. Das\(^1\)

\(^1\)Pennsylvania State University  \(^2\)College of William and Mary  
\(^3\)Advanced Micro Devices, Inc.  \(^4\)Intel Labs  \(^5\)ETH Zürich  \(^6\)Carnegie Mellon University
Challenge: Coherence for Hybrid CPU-PIM Apps

Traditional coherence
No coherence overhead
How to Maintain Coherence?

- 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
How to Maintain Coherence?


CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators

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

†Carnegie Mellon University
‡ETH Zürich
◊Simon Fraser University
§Samsung Semiconductor, Inc.
PTSUTNB
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
How to Design Data Structures for PIM?


Concurrent Data Structures for Near-Memory Computing

Zhiyu Liu
Computer Science Department
Brown University
zhiyu.liu@brown.edu

Maurice Herlihy
Computer Science Department
Brown University
mph@cs.brown.edu

Irina Calciu
VMware Research Group
icalciu@vmware.com

Onur Mutlu
Computer Science Department
ETH Zürich
onur.mutlu@inf.ethz.ch
Simulation Infrastructures for PIM

- **Ramulator** extended for PIM
  - Flexible and extensible DRAM simulator
  - Can model many different memory standards and proposals
  - [https://github.com/CMU-SAFARI/ramulator](https://github.com/CMU-SAFARI/ramulator)
An FPGA-based Test-bed for PIM?


- Flexible
- Easy to Use (C++ API)
- Open-source

`github.com/CMU-SAFARI/SoftMC`
Simulation Infrastructures for PIM (in SSDs)

- Arash Tavakkol, Juan Gomez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu, "MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices"


[Slides (pptx) (pdf)]
[Source Code]
New Applications and Use Cases for PIM

Jeremie S. Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu,
"GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies"

Proceedings of the 16th Asia Pacific Bioinformatics Conference (APBC), Yokohama, Japan, January 2018.
arxiv.org Version (pdf)

GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies

Jeremie S. Kim¹,⁶*, Damla Senol Cali¹, Hongyi Xin², Donghyuk Lee³, Saugata Ghose¹, Mohammed Alser⁴, Hasan Hassan⁶, Oguz Ergin⁵, Can Alkan⁴* and Onur Mutlu⁶,¹*

From The Sixteenth Asia Pacific Bioinformatics Conference 2018
Yokohama, Japan. 15-17 January 2018
Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks

Amirali Boroumand

Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, Onur Mutlu

SAFARI
Carnegie Mellon
Google
Samsung
Seoul National University
ETH Zürich
Genome Read In-Memory (GRIM) Filter:
Fast Seed Location Filtering in DNA Read Mapping using Processing-in-Memory Technologies

Jeremie Kim,
Damla Senol, Hongyi Xin, Donghyuk Lee,
Saugata Ghose, Mohammed Alser, Hasan Hassan,
Oguz Ergin, Can Alkan, and Onur Mutlu
Executive Summary

- **Genome Read Mapping** is a very important problem and is the first step in many types of genomic analysis
  - Could lead to improved health care, medicine, quality of life

- Read mapping is an **approximate string matching** problem
  - Find the best fit of 100 character strings into a 3 billion character dictionary
  - **Alignment** is currently the best method for determining the similarity between two strings, but is very expensive

- We propose an in-memory processing algorithm **GRIM-Filter** for accelerating read mapping, by reducing the number of required alignments

- We implement GRIM-Filter using **in-memory processing** within 3D-stacked memory and show up to **3.7x speedup**.
The layout of bit vectors in a bank enables filtering many bins in parallel

- Customized logic for accumulation and comparison per genome segment
  - Low area overhead, simple implementation
GRIM-Filter Performance

1.8x-3.7x performance benefit across real data sets
GRIM-Filter False Positive Rate

False Positive Rate (%)

Benchmarks and their False Positive Rates

5.6x-6.4x False Positive reduction across real data sets
Conclusions

- We propose an **in memory filter algorithm** to **accelerate end-to-end genome read mapping** by reducing the number of required alignments.

- Compared to the previous best filter:
  - We observed **1.8x-3.7x speedup**
  - We observed **5.6x-6.4x fewer false positives**

- **GRIM-Filter is a universal filter** that can be applied to any genome read mapper.
In-Memory DNA Sequence Analysis

- Jeremie S. Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies"
  
  *BMC Genomics*, 2018.
  
  *Proceedings of the 16th Asia Pacific Bioinformatics Conference (APBC)*, Yokohama, Japan, January 2018.
  

---

GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies

Jeremie S. Kim¹,⁶*, Damla Senol Cali¹, Hongyi Xin², Donghyuk Lee³, Saugata Ghose¹, Mohammed Alser⁴, Hasan Hassan⁶, Oguz Ergin⁵, Can Alkan⁴* and Onur Mutlu⁶,¹*

*From The Sixteenth Asia Pacific Bioinformatics Conference 2018
Yokohama, Japan. 15-17 January 2018*
Processing Data Where It Makes Sense: Enabling In-Memory Computation

Onur Mutlu$^{a,b}$, Saugata Ghose$^b$, Juan Gómez-Luna$^a$, Rachata Ausavarungnirun$^{b,c}$

$^a$ETH Zürich
$^b$Carnegie Mellon University
$^c$King Mongkut’s University of Technology North Bangkok

Onur Mutlu, Saugata Ghose, Juan Gomez-Luna, and Rachata Ausavarungnirun, "Processing Data Where It Makes Sense: Enabling In-Memory Computation"

Invited paper in Microprocessors and Microsystems (MICPRO), June 2019.

[arXiv version]

Enabling the Paradigm Shift
Computer Architecture Today

- You can revolutionize the way computers are built, if you understand both the hardware and the software (and change each accordingly)

- You can invent new paradigms for computation, communication, and storage

- Recommended book: Thomas Kuhn, “The Structure of Scientific Revolutions” (1962)
  - Pre-paradigm science: no clear consensus in the field
  - Normal science: dominant theory used to explain/improve things (business as usual); exceptions considered anomalies
  - Revolutionary science: underlying assumptions re-examined
Computer Architecture Today

- You can revolutionize the way computers are built, if you understand both the hardware and the software (and change each accordingly).
- You can invent new paradigms for computation, communication, and storage.
- Recommended book: Thomas Kuhn, "The Structure of Scientific Revolutions" (1962)
  - Pre-paradigm science: no clear consensus in the field
  - Normal science: dominant theory used to explain/improve things (business as usual); exceptions considered anomalies
  - Revolutionary science: underlying assumptions re-examined
Agenda

- Major Trends Affecting Main Memory
- The Need for Intelligent Memory Controllers
  - Bottom Up: Push from Circuits and Devices
  - Top Down: Pull from Systems and Applications
- Processing in Memory: Two Directions
  - Minimally Changing Memory Chips
  - Exploiting 3D-Stacked Memory
- How to Enable Adoption of Processing in Memory
- Conclusion
Maslow’s Hierarchy of Needs, A Third Time


Source: https://www.simplypsychology.org/maslow.html
Challenge and Opportunity for Future

Fundamentally High-Performance (Data-Centric) Computing Architectures
Challenge and Opportunity for Future

Fundamentally Energy-Efficient (Data-Centric) Computing Architectures
Challenge and Opportunity for Future

Fundamentally Low-Latency (Data-Centric) Computing Architectures
Challenge and Opportunity for Future Computing Architectures with Minimal Data Movement
PIM: Concluding Remarks
A Quote from A Famous Architect

- “architecture [...] based upon principle, and not upon precedent”
Precedent-Based Design?

- “architecture [...] based upon principle, and not upon precedent”
Principled Design

- “architecture [...] based upon principle, and not upon precedent”
The Overarching Principle

**Organic architecture**

From Wikipedia, the free encyclopedia

*Organic architecture* is a philosophy of architecture which promotes harmony between human habitation and the natural world through design approaches so sympathetic and well integrated with its site, that buildings, furnishings, and surroundings become part of a unified, interrelated composition.

A well-known example of organic architecture is *Fallingwater*, the residence Frank Lloyd Wright designed for the Kaufmann family in rural Pennsylvania. Wright had many choices to locate a home on this large site, but chose to place the home directly over the waterfall and creek creating a close, yet noisy dialog with the rushing water and the steep site. The horizontal striations of stone masonry with daring *cantilevers* of colored beige concrete blend with native rock outcroppings and the wooded environment.
Another Example: Precedent-Based Design

Source: http://cookiemagik.deviantart.com/art/Train-station-207266944
Principled Design

Source: By Toni_V, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=4087256
Another Principled Design
Another Principled Design
Principle Applied to Another Structure


Source: By Forgemind ArchiMedia - Flickr: IMG_2489 JPG, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=31493356

Source: https://en.wikipedia.org/wiki/Santiago_Calatrava
The Overarching Principle

Zoomorphic architecture

From Wikipedia, the free encyclopedia

Zoomorphic architecture is the practice of using animal forms as the inspirational basis and blueprint for architectural design. "While animal forms have always played a role adding some of the deepest layers of meaning in architecture, it is now becoming evident that a new strand of biomorphism is emerging where the meaning derives not from any specific representation but from a more general allusion to biological processes."

[1]

Some well-known examples of Zoomorphic architecture can be found in the TWA Flight Center building in New York City, by Eero Saarinen, or the Milwaukee Art Museum by Santiago Calatrava, both inspired by the form of a bird’s wings. [3]
Overarching Principle for Computing?
Concluding Remarks

- It is time to design principled system architectures to solve the memory problem.

- Design complete systems to be balanced, high-performance, and energy-efficient, i.e., data-centric (or memory-centric).

- Enable computation capability inside and close to memory.

- This can:
  - Lead to orders-of-magnitude improvements.
  - Enable new applications & computing platforms.
  - Enable better understanding of nature.
  - ...

The Future of Processing in Memory is Bright

- Regardless of challenges
  - in underlying technology and overlying problems/requirements

Can enable:
- Orders of magnitude improvements
- New applications and computing systems

Yet, we have to
- Think across the stack
- Design enabling systems
We Need to Revisit the Entire Stack

We can get there step by step
If In Doubt, See Other Doubtful Technologies

- A very “doubtful” emerging technology
  - for at least two decades

*Proceedings of the IEEE, Sept. 2017*

Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives

This paper reviews the most recent advances in solid-state drive (SSD) error characterization, mitigation, and data recovery techniques to improve both SSD’s reliability and lifetime.

By Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu

https://arxiv.org/pdf/1706.08642
Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarunngirun, "Processing Data Where It Makes Sense: Enabling In-Memory Computation"


[arXiv version]

GRIM-Filter:
Fast seed location filtering in DNA read mapping using processing-in-memory technologies

Jeremie S. Kim,
Damla Senol Cali, Hongyi Xin, Donghyuk Lee,
Saugata Ghose, Mohammed Alser, Hasan Hassan,
Oguz Ergin, Can Alkan, and Onur Mutlu
Genome Read Mapping is a very important problem and is the first step in genome analysis.

Read Mapping is an approximate string matching problem:
- Find the best fit of 100 character strings into a 3 billion character dictionary
- Alignment is currently the best method for determining the similarity between two strings, but is very expensive.

We propose an algorithm called GRIM-Filter:
- Accelerates read mapping by reducing the number of required alignments
- GRIM-Filter can be accelerated using processing-in-memory
  - Adds simple logic into 3D-Stacked memory
  - Uses high internal memory bandwidth to perform parallel filtering
- GRIM-Filter with processing-in-memory delivers a 3.7x speedup.
# GRIM-Filter Outline

<table>
<thead>
<tr>
<th>1. Motivation and Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Background Read Mappers</td>
</tr>
<tr>
<td>a. Hash Table Based</td>
</tr>
<tr>
<td>b. Hash Table Based with Filter</td>
</tr>
<tr>
<td>3. Our Proposal: GRIM-Filter</td>
</tr>
<tr>
<td>4. Mapping GRIM-Filter to 3D-Stacked Memory</td>
</tr>
<tr>
<td>5. Results</td>
</tr>
<tr>
<td>6. Conclusion</td>
</tr>
</tbody>
</table>
Motivation and Goal

- **Sequencing**: determine the [A,C,G,T] series in DNA strand

- Today’s machines sequence short strands (**reads**)
  - Reads are on the order of 100 – 20k base pairs (**bp**)
  - The human genome is approximately 3 billion bp

- Therefore genomes are cut into reads, which are sequenced independently, and then reconstructed
  - **Read mapping** is the first step in analyzing someone’s genome to detect predispositions to diseases, personalize medicine, etc.

- **Goal**: We want to **accelerate** end-to-end performance of **read mapping**
GRIM-Filter Outline

1. Motivation and Goal

2. Background: Read Mappers
   - a. Hash Table Based
   - b. Hash Table Based with Filter

3. Our Proposal: GRIM-Filter

4. Mapping GRIM-Filter to 3D-Stacked Memory

5. Results

6. Conclusion
Background: Read Mappers

We now have **sequenced reads** and want a **full genome** via Read Mapping.

We map **reads** to a known **reference genome** (>99.9% similarity across humans) with some minor errors allowed.

Because of high similarity, long sequences in **reads** perfectly match in the **reference genome**.

We can use a hash table to help quickly map the **reads**!
# GRIM-Filter Outline

<table>
<thead>
<tr>
<th>1. Motivation and Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Background: Read Mappers</td>
</tr>
<tr>
<td>a. Hash Table Based</td>
</tr>
<tr>
<td>b. Hash Table Based with Filter</td>
</tr>
<tr>
<td>3. Our Proposal: GRIM-Filter</td>
</tr>
<tr>
<td>4. Mapping GRIM-Filter to 3D-Stacked Memory</td>
</tr>
<tr>
<td>5. Results</td>
</tr>
<tr>
<td>6. Conclusion</td>
</tr>
</tbody>
</table>
Generating Hash Tables

To map any reads, generate a hash table per reference genome.

We can query the table with substrings from reads to quickly find a list of possible mapping locations.
Hash Tables in Read Mapping

Read Sequence (100 bp)

99.9% of locations result in a mismatch

We want to filter these out so we do not waste time trying to align them
Location Filtering

- **Alignment** is expensive and requires the use of $O(n^2)$ dynamic programming algorithm
  - We need to align millions to billions of reads

- Modern read mappers reduce the time spent on alignment for increased performance. Can be done in two ways:
  1. Optimize the algorithm for alignment
  2. Reduce the number of alignments necessary by filtering out mismatches quickly

- Both methods are used by mappers today, but filtering has replaced alignment as the bottleneck [Xin+, BMC Genomics 2013]
GRIM-Filter Outline

1. Motivation and Goal

2. Background: Read Mappers
   a. Hash Table Based
   b. Hash Table Based with Filter

3. Our Proposal: GRIM-Filter

4. Mapping GRIM-Filter to 3D-Stacked Memory

5. Results

6. Conclusion
1. Motivation and Goal

2. Background: Read Mappers
   a. Hash Table Based
   b. Hash Table Based with Filter

3. Our Proposal: GRIM-Filter

4. Mapping GRIM-Filter to 3D-Stacked Memory

5. Results

6. Conclusion
Our Proposal: GRIM-Filter

1. **Data Structures: Bins & Bitvectors**
2. Checking a Bin
3. Integrating GRIM-Filter into a Mapper
GRIM-Filter: Bins

- We partition the genome into large sequences (bins).

Represent each bin with a **bitvector** that holds the occurrence of all permutations of a small string (**token**) in the bin.

To account for matches that straddle bins, we employ overlapping bins.

- A read will now always completely fall within a single bin.

Bitvector:

- **AAAAA**
- **AAAAC**
- **AAAAT**
- **...**
- **CCCCC**
- **CCCCT**
- **CCCCG**
- **...**
- **GGGGG**

- **AAAAA** exists in bin x
- **CCCCT** doesn’t exist in bin x
GRIM-Filter: Bitvectors

...CGTGA... AGT... C...

Bin x

Bin x Bitvector

AAAAA 0
...
CGTGA
...
TGAGT
...
GAGTC
...
GTGAG
...
GRIM-Filter: Bitvectors

Storing all bitvectors requires $4^n \times t$ bits in memory, where $t = \text{number of bins}$.

For bin size $\sim$200, and $n = 5$, memory footprint $\sim$3.8 GB
GRIM-Filter: Checking a Bin

How GRIM-Filter determines whether to discard potential match locations in a given bin prior to alignment

1. Get tokens
2. Read bitvector for bin_num(x)
3. Match tokens to bitvector
4. Sum
5. Compare

INPUT: Read Sequence $r$
GAACCTGGAGTCTA ... CGAG

≥ Threshold?

Send to Read Mapper for Sequence Alignment

Compare

Discard

NO

YES
Our Proposal: GRIM-Filter

1. Data Structures: Bins & Bitvectors
2. Checking a Bin
3. Integrating GRIM-Filter into a Mapper
Our Proposal: GRIM-Filter

1. Data Structures: Bins & Bitvectors
2. Checking a Bin
3. Integrating GRIM-Filter into a Mapper
Our Proposal: GRIM-Filter

1. Data Structures: Bins & Bitvectors
2. Checking a Bin
3. Integrating GRIM-Filter into a Mapper
Integrating GRIM-Filter into a Read Mapper

**INPUT**: Read Sequence

```
GAACCTTGCAG ... GTATT
```

1. **GRIM-Filter**: Filter Bitmask Generator

```
...0001010 ...011010...
```

Seed Location Filter Bitmask

**INPUT**: All Potential Seed Locations

```
...020128 ...020131 ...414415...
```

2. **GRIM-Filter**: Seed Location Checker

```
...0001010 ...01101001...
```

Seed Location Filter Bitmask

**INPUT**: All Potential Seed Locations

```
...020128 ...020131 ...414415...
```

**REFERENCE SEGMENT STORAGE**

```
reference segment @ 020131
```

```
reference segment @ 414415
```

**OUTPUT**: Correct Mappings

**SAFARI**
# GRIM-Filter Outline

1. Motivation and Goal

2. Background: Read Mappers
   - a. Hash Table Based
   - b. Hash Table Based with Filter

3. Our Proposal: GRIM-Filter

4. Mapping GRIM-Filter to 3D-Stacked Memory

5. Results

6. Conclusion
Key Properties of GRIM-Filter

1. **Simple Operations:**
   - To check a given bin, find the **sum** of all bits corresponding to each token in the read
   - **Compare** against threshold to determine whether to align

2. **Highly Parallel:** Each bin is operated on independently and there are many many bins

3. **Memory Bound:** Given the frequent accesses to the large bitvectors, we find that GRIM-Filter is memory bound

These properties together make GRIM-Filter a good algorithm to be run in 3D-Stacked DRAM
Hash Tables in Read Mapping

Read Sequence (100 bp)

Hash Table

Reference Genome

Filter

37  140  894  1203  1564

False Negative
3D-Stacked Memory

- 3D-Stacked DRAM architecture has extremely high bandwidth as well as a stacked customizable logic layer
  - Logic Layer enables Processing-in-Memory, offloading computation to this layer and alleviating the memory bus
  - Embed GRIM-Filter operations into DRAM logic layer and appropriately distribute bitvectors throughout memory
3D-Stacked Memory

- 3D-Stacked DRAM architecture has **bandwidth** as well as a stacked customizable **logic layer**.
  - Logic Layer enables **Processing in Memory**, offloading computation to this layer and alleviating the memory bus.
  - Embed **GRIM**-Filter operations into DRAM logic layer and appropriately distribute bitvectors throughout memory.

3D-Stacked Memory

Micron’s HMC

Micron has working demonstration components

http://images.anandtech.com/doci/9266/HBMCar_678x452.jpg

Each DRAM layer is organized as an array of **banks**
- A **bank** is an array of cells with a row buffer to transfer data

The layout of bitvectors in a bank enables filtering many bins in parallel
GRIM-Filter in 3D-Stacked DRAM

- Customized logic for accumulation and comparison per genome segment
  - Low area overhead, simple implementation
  - For HBM2, we use 4096 incrementer LUTs, 7-bit counters, and comparators in logic layer

Details are in the paper
## GRIM-Filter Outline

1. Motivation and Goal

2. Background: Read Mappers
   - a. Hash Table Based
   - b. Hash Table Based with Filter

3. Our Proposal: GRIM-Filter

4. Mapping GRIM-Filter to 3D-Stacked Memory

5. Results

6. Conclusion
Methodology

- Performance simulated using an in-house 3D-Stacked DRAM simulator
- Evaluate 10 real read data sets (From the 1000 Genomes Project)
  - Each data set consists of 4 million reads of length 100
- Evaluate two key metrics
  - Performance
  - False negative rate
    - The fraction of locations that pass the filter but result in a mismatch
- Compare against a state-of-the-art filter, FastHASH [Xin+., BMC Genomics 2013] when using mrFAST, but GRIM-Filter can be used with ANY read mapper
GRIM-Filter Performance

Benchmarks and their Execution Times

<table>
<thead>
<tr>
<th>Time (x1000 seconds)</th>
<th>FastHASH filter</th>
<th>GRIM-Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR240726-1</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>ERR240726-2</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>ERR240727-1</td>
<td>25</td>
<td>12.5</td>
</tr>
<tr>
<td>ERR240727-2</td>
<td>35</td>
<td>17.5</td>
</tr>
<tr>
<td>ERR240728-1</td>
<td>22</td>
<td>11</td>
</tr>
<tr>
<td>ERR240728-2</td>
<td>33</td>
<td>16.5</td>
</tr>
<tr>
<td>ERR240729-1</td>
<td>24</td>
<td>12</td>
</tr>
<tr>
<td>ERR240729-2</td>
<td>34</td>
<td>17</td>
</tr>
<tr>
<td>ERR240730-1</td>
<td>36</td>
<td>18</td>
</tr>
<tr>
<td>ERR240730-2</td>
<td>38</td>
<td>19</td>
</tr>
<tr>
<td>Average</td>
<td>28</td>
<td>14</td>
</tr>
</tbody>
</table>

Sequence Alignment Error Tolerance ($e$) $e = 0.05$

1.8x-3.7x performance benefit across real data sets

2.1x average performance benefit

GRIM-Filter gets performance due to its hardware-software co-design
GRIM-Filter False Negative Rate

Benchmarks and their False Negative Rates

- FastHASH filter
- GRIM-Filter

Sequence Alignment Error Tolerance \(e\)

\[ e = 0.05 \]

5.6x-6.4x False Negative reduction across real data sets

6.0x average reduction in False Negative Rate

GRIM-Filter utilizes more information available in the read to filter
Other Results in the Paper

- Sensitivity of execution time and false negative rates to error tolerance of string matching
- Read mapper execution time breakdown
- Sensitivity studies on the filter
  - Token Size
  - Bin Size
  - Error Tolerance
1. Motivation and Goal

2. Background: Read Mappers
   a. Hash Table Based
   b. Hash Table Based with Filter

3. Our Proposal: GRIM-Filter

4. Mapping GRIM-Filter to 3D-Stacked Memory

5. Results

6. Conclusion
Conclusion

We propose an in-memory filtering algorithm to accelerate end-to-end read mapping by reducing the number of required alignments.

Key ideas:
- Introduce a new representation of coarse-grained segments of the reference genome.
- Use massively-parallel in-memory operations to identify read presence within each coarse-grained segment.

Key contributions and results:
- Customized filtering algorithm for 3D-Stacked DRAM.
- Compared to the previous best filter:
  - We observed 1.8x-3.7x read mapping speedup.
  - We observed 5.6x-6.4x fewer false negatives.

GRIM-Filter is a universal filter that can be applied to any read mapper.
GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies

Jeremie S. Kim,
Damla Senol Cali, Hongyi Xin, Donghyuk Lee,
Saugata Ghose, Mohammed Alser, Hasan Hassan,
Oguz Ergin, Can Alkan, and Onur Mutlu

Systems@ETH Zürich
Carnegie Mellon
Eth Zürich
SAFARI
Bilkent University
TOBB University of Economics and Technology
In-Memory DNA Sequence Analysis

- Jeremie S. Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies"
  
  *BMC Genomics*, 2018.

Proceedings of the *16th Asia Pacific Bioinformatics Conference* (APBC), Yokohama, Japan, January 2018.

*arxiv.org Version (pdf)*

---

**GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies**

Jeremie S. Kim¹,6*, Damla Senol Cali¹, Hongyi Xin², Donghyuk Lee³, Saugata Ghose¹, Mohammed Alser⁴, Hasan Hassan⁶, Oguz Ergin⁵, Can Alkan⁴* and Onur Mutlu⁶,1*  

*From The Sixteenth Asia Pacific Bioinformatics Conference 2018*  
Yokohama, Japan. 15-17 January 2018
LazyPIM
An Efficient Cache Coherence Mechanism for Processing In Memory

Amirali Boroumand

"LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory",
IEEE CAL 2016. (Preliminary version)
LazyPIM Summary

• Cache Coherence is a major system challenge for PIM
  – Conventional cache coherence makes PIM programming easy but loses a significant portion of PIM benefits

• Observation:
  – Significant amount of sharing between PIM cores and CPU cores in many important data-intensive applications
  – Efficient handling of coherence is critical to retain PIM benefits

• LazyPIM
  – Key idea: use speculation to avoid coherence lookups during PIM core execution and compressed signatures to verify correctness after PIM core is done
  – Improves performance by 19.8% and energy by 18% vs. best previous
  – Comes within 4.4% and 9.8% of ideal PIM energy and performance

• We believe LazyPIM can enable new applications that benefit from fine-grained sharing between CPU and PIM
PIM Coherence

• A Major System Challenge for PIM: Coherence

Need a coherence mechanism to ensure correctness!
PIM Coherence

- **Potential solution: Conventional coherence protocols**
  - We can treat PIM cores as *additional independent cores*
  - Use *conventional coherence protocol* to make them coherent with the CPUs

Conventional coherence is **impractical**: large number of coherence messages over off-chip channel

- ✔️ Simplifies PIM programming model
- ✗ Generates a large amount of off-chip coherence traffic
- ✗ Eliminates on average **72.4%** of Ideal PIM energy improvement
Goal and Key Idea

• Our goal is to develop a cache coherence mechanism that:
  1) Maintains the logical behavior of conventional cache coherence protocols to simplify PIM programming model
  2) Retains the large performance and energy benefits of PIM

• Our key idea is
  1) Avoid coherence lookups during PIM core execution
  2) Batch lookups in compressed signatures and use them to verify correctness after PIM core finishes
Background

Prior Approaches to PIM Coherence
Prior Approaches to PIM Coherence

- There are many recent proposals on PIM
  - Primarily focus on the design of compute unit within the logic layer

- Prior works employ other approaches than conventional coherence protocol
  - Marking PIM-data as Non-cacheable
    - They no longer need to deal with coherence
  - Coarse-grained coherence
    - Tracks coherence at a larger granularity than a single cache line
    - Does not transfer permission while PIM is working
    - No concurrent access from the CPU and PIM
Prior Approaches to PIM Coherence

• Prior works proposed coherence mechanisms assuming:
  – Entire application could be offloaded to PIM core \(\rightarrow\) **Almost zero sharing** between PIM and CPU
  – Only **limited** communication happens between CPU and PIM

**Observation:** These assumptions **do not hold** for many important data-intensive applications that benefit from PIM
Motivation

Applications with Data Sharing
Application Analysis for PIM

• An application benefits from PIM when we offload its memory-intensive parts that:
  – Generate a lot of data movement
  – Have poor cache locality
  – Contribute to a large portion of execution time

• Parts of the application that are compute-intensive or cache friendly should remain on the CPU
  – To benefit from larger and sophisticated cores with larger caches
Example: Hybrid In-Memory Database

Transactional Threads (CPU Friendly)

Analytical Threads (PIM Friendly)

Hybrid Database

Data Sharing

Ideal PIM vs. CPU-only:
1.93x Speedup
68% reduction in energy
Applications with High Data Sharing

• Our application analysis shows that:
  – Some portions of the applications perform better on CPUs
  – These portions often access the same region of data as the PIM cores

• Based on this observation, we can conclude that:
  – There are important data-intensive applications that have strong potential for PIM and show significant data sharing between the CPU and PIM
Let’s see how prior approaches work for these applications
Non-Cacheable

- Generates a large number of off-chip accesses
- Significantly hurts CPU threads’ performance
Motivation: Summary

• **Conventional cache coherence** loses a significant portion of PIM benefits

• Prior works use other approaches to avoid those costs
  — Their assumption: *Zero* or *a limited* amount of sharing

• **We observe that those assumptions do not hold** for a number of important data-intensive applications
  — Using prior approaches eliminates a significant portion of PIM benefits

• We want to get the best of both worlds
  1) Maintain the **logical behavior** of conventional cache coherence
  2) **Retain** the large **performance and energy benefits** of PIM
LazyPIM
Baseline PIM Architecture
Our Proposal

- **LazyPIM:**
  - Lets PIM cores use *speculation to avoid* coherence lookups *during execution*
  - Uses *compressed signatures* to batch the lookups and verify correctness *after* the PIM core completes
LazyPIM High-level Operation

1) CPU portion execution

2) Offload PIM kernel

3) PIM portion execution

4) Send PIM Signatures

5) Conflict Check

6) Commit or Rollback

No Coherence
How LazyPIM Avoids Pitfalls of Prior Approaches

• **Conventional Coherence (Fine-grained)**
  - ✗ Generates a large amount of off-chip coherence traffic *for every miss*
  - ✔ LazyPIM only sends a compressed signature after PIM cores finishes

• **Coarse-grained Coherence**
  - ✗ Unnecessarily flushes a large amount of data
  - ✔ LazyPIM performs only the necessary flushes
  - ✗ Causes Thread Serialization
  - ✔ LazyPIM enables concurrent execution of the CPUs and PIM cores

• **Non-Cacheable**
  - ✗ A large number of off-chip accesses hurting CPU threads’ performance
  - ✔ LazyPIM allows CPU threads to use caches
Coarse-Grained Coherence

• Need to get coherence permission for the entire region
  – Needs to **flush** every dirty data within that region to transfer permission

× Unnecessarily flushes a large amount of data in **pointer-based data structure**

• **Does not allow concurrent accesses**
  – **Blocks CPUs** accessing PIM-data during PIM execution

× **Coarse-grained locks frequently cause thread serialization**
How we define conflicts in LazyPIM?
Conflicts

1) PIM Read and Processor Write: **Conflict**

2) Processor Read and PIM Write: **No Conflict**

3) Processor Write and PIM Write: **No Conflict**

1) Offload PIM kernel

2) Send PIM signatures

5) Commit PIM data
Architecture Support
LazyPIM Architecture

• How does LazyPIM support **speculative execution**?
• How does LazyPIM implement **signatures**?
• How does LazyPIM **handle conflicts**?
Tracking speculative updates

- One-bit flag per cache line to mark all data updates as speculative
Tracking potential conflicts

- The CPU records all dirty cache lines and writes in the PIM data region in the `CPUWriteSet`.

Tracking memory accesses

- The `PIMReadSet` and `PIMWriteSet` are updated for every read and write by the PIM core.
Bloom filter based signature has two major benefits:

- Allows us to easily perform conflict detection
- Allows for a large number of addresses to be stored within a fixed-length register
Handling Conflicts

If **conflict** happens:
- The CPU flushes the dirty cache lines that match addresses in the PIMReadSet
- PIM core invalidates all speculative cache lines
- Signatures are erased and PIM core restarts execution

If **no conflicts**:
- Any clean cache lines in the CPU that match an address in the PIMWriteSet are invalidated
- PIM core commits speculative updates
Evaluation
Evaluation Methodology

• **Simulator**
  - Gem5 full system simulator

• **System Configuration:**
  - **Processor**
    - 4-16 Cores, 8 wide issue, 2GHz Frequency
    - L1 I/D Cache: 64KB private, 4-way associative, 64B Block
    - L2 Cache: 2MB shared, 8-way associative, 64B Blocks
    - Cache Coherence Protocol: MESI
  - **PIM**
    - 4-16 Cores, 1 wide issue, 2GHz Frequency
    - L1 I/D Cache: 64KB private, 4-way associative, 64B Block
    - Cache Coherence Protocol: MESI
  - **3D-stacked Memory**
    - One 4GB Cube, 16 Vaults per cube
Applications

- **Ligra**
  - Lightweight multithreaded graph processing for shared memory system
  - We used three Ligra graph applications
    - PageRank
    - Radii
    - Connected Components
  - Input graphs constructed from real-world network datasets:
    - arXiv General Relativity (5K nodes, 14K edges)
    - peer-to-peer Gnutella25 (22K nodes, 54K edges).
    - Enron email communication network (36K nodes, 183K edges)

- **IMDB**
  - In-house prototype of an in-memory database (IMDB)
  - Capable of running both transactional queries and analytical queries on the same database tables (HTAP workload)
  - 32K transactions, 128/256 analytical queries
Speedup with 16 Threads

- **CPU-only**
- **FG**
- **CG**
- **NC**
- **LazyPIM**
- **Ideal-PIM**

FG loses a significant portion of Ideal-PIM’s improvement.

LazyPIM consistently retains most of Ideal-PIM’s benefits, coming within 9.8% of the Ideal-PIM performance.
Energy with 16 threads

- **NC** suffers greatly from the *large number of accesses to DRAM*
- **Interconnect** and **DRAM** energy increase by *3.1x* and *4.5x*

**LazyPIM** significantly reduces energy consumption and comes within *4.4%* of **Ideal-PIM**
Conclusion
Conclusion

• **Cache Coherence is a major system challenge for PIM**
  – Conventional cache coherence makes PIM programming easy but loses a significant portion of PIM benefits

• **Observation:**
  – Significant amount of sharing between PIM cores and CPU cores in many important data-intensive applications
  – Efficient handling of coherence is critical to retain PIM benefits

• **LazyPIM**
  – **Key idea:** use speculation to avoid coherence lookups during PIM core execution and compressed signatures to verify correctness after PIM core is done
  – Improves performance by 19.8% and energy by 18% vs. best previous
  – Comes within 4.4% and 9.8% of ideal PIM energy and performance

• **We believe LazyPIM can enable new applications that benefit from fine-grained sharing between CPU and PIM**
LazyPIM
An Efficient Cache Coherence Mechanism for Processing In Memory

Amirali Boroumand

"LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory",
IEEE CAL 2016. (Preliminary version)
Efficient Automatic Data Coherence Support

- 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
End of Backup Slides