iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture

Peng Gu*1, Xinfeng Xie*1, Yufei Ding1
Guoyang Chen2, Weifeng Zhang2, Dimin Niu2, Yuan Xie1,2

1University of California, Santa Barbara
2Alibaba Group

*co-primary first authors
Image Processing: Application Scenarios

- Smartphone Camera
- Traffic Camera
- Medical Imaging Equipment
- Satellite Imaging
- Data Center, Work Station
- Machine Learning
- Traffic Analysis
- Medical Image Processing
- Geographic Information System

http://www.nectec.or.th/2008/r-d/img.html
https://www.surrey.ac.uk/centre-vision-speech-signal-processing/research/m-lab-biomedical-imaging-and-processing
https://clarklabs.org/tersset/idrisi-image-processing/
Image Processing: Application Scenarios

Domain Programmability
High Performance
High Energy Efficiency

Image Processing Accelerator

Smartphone Camera
Traffic Camera
Medical Imaging Equipment
Satellite Imaging

Data Center, Work Station

Machine Learning
Traffic Analysis
Medical Image Processing
Geographic Information System

Scalable and Energy-Efficient Architecture Lab (SEAL)

http://www.nectec.or.th/2008/r-d/img.html
https://www.surrey.ac.uk/centre-vision-speech-signal-processing/research/m-lab-biomedical-imaging-and-processing
https://clarklabs.org/terrsset/idrisi-image-processing/
Image Processing: Characteristics

- Image Processing algorithms consist of pipeline stages that are both *wide* and *heterogeneous*
  - Each stage is *wide* – Example: Image Blur
Image Processing: Characteristics

- Image Processing algorithms consist of pipeline stages that are both **wide** and **heterogeneous**
  - Each stage is **wide** – Example: Image Blur

1. Memory-level parallelism among pixels
2. Large size of intermediate data
3. Low arithmetic density operations
Image Processing: Characteristics

• Image Processing algorithms consist of pipeline stages that are both *wide* and *heterogeneous*
  
  • Overall pipelines are *heterogeneous* – Example: Local Laplacian Filter

Image Processing: Characteristics

- Image Processing Algorithms consist of pipeline stages that are both wide and heterogeneous

Image processing workloads have high memory bandwidth demand:

- Software: low temporal reuse due to
  - (1) low arithmetic density
  - (2) difficulty of pipeline fusion
- Hardware: on-chip cache cannot hold all intermediate data
Motivation Data: Memory Bandwidth Bottleneck

- On average: **57.55%** memory utilization v.s. **3.43%** ALU utilization
  - Benchmark: single-stage / multi-stage kernels
  - Configuration: Halide toolchain on a Tesla V100 GPU
Motivation Data: Memory Bandwidth Bottleneck

<table>
<thead>
<tr>
<th>Single-stage kernels</th>
<th>Multi-stage kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU utilization: 2.85%</td>
<td>ALU utilization: 4.53%</td>
</tr>
<tr>
<td>DRAM utilization: 58.80%</td>
<td>DRAM utilization: 55.73%</td>
</tr>
</tbody>
</table>

![Graph showing bandwidth and utilization for different kernels]

- **DRAM bandwidth**: Various values for different kernels.
- **DRAM utilization**: Various values for different kernels.
- **ALU utilization**: Various values for different kernels.
Motivation Data: Memory Bandwidth Bottleneck

Pipeline optimization does not change memory-bound behavior of image processing workloads on GPU.
GPU: Bandwidth Scaling Challenge

**GPU provides the highest memory bandwidth**: High Bandwidth Memory (HBM) provides large bandwidth by 3D die-stacking technology.
GPU: Bandwidth Scaling Challenge

Vertical interconnect: Through Silicon Vias (TSVs)

Memory bandwidth wall: memory bandwidth cannot scale with computation throughput
- Off-chip I/O (Interposer Interconnects)
- TSV I/O
GPU: Bandwidth Scaling Challenge

- Raw memory bandwidth = (the number of I/Os) \times (data rate)

- Increasing the number of I/Os is difficult under tight area budget
  - Limited off-chip pins
  - TSVs already consumes ~18.8% DRAM die area for the current HBM2
GPU: Bandwidth Scaling Challenge

- Raw memory bandwidth = \((\text{the number of I/Os}) \times (\text{data rate})\)
- Increasing data rate will have signal integrity issues and increase power consumption as well
Summary

• Image processing is important in many application domains
• GPU suffers from memory bandwidth bottleneck:
  • Software: image processing pipelines are wide and heterogeneous
  • Hardware: GPU has bandwidth scaling challenges

How to design a programmable image processing accelerator to provide more memory bandwidth?
3D-Stacking Processing-in-memory (PIM) Architecture

• Key idea:
  • Integrate computation logic closer to physical memory in order to increase memory bandwidth and reduce data movement energy
3D-Stacking Processing-in-memory (PIM) Architecture

- Key idea:
  - Integrate computation logic closer to physical memory in order to increase memory bandwidth and reduce data movement energy.

- Diagram:
  - GPU + 3D memory
  - Process-on-base-die
  - Off-chip I/O bound
  - Move core to base die
3D-Stacking Processing-in-memory (PIM) Architecture

• Key idea:
  • Integrate computation logic **closer to** physical memory in order to increase memory bandwidth and reduce data movement energy
Challenges for near-bank architecture

Complex computation and memory access patterns

Resource constraints
Challenges for near-bank architecture

Complex computation and memory access patterns

Lightweight programmable architecture for image processing domain
Challenges for near-bank architecture

Complex computation and memory access patterns

Concise yet powerful Instruction Set Architecture (ISA)

Lightweight programmable architecture for image processing domain
Challenges for near-bank architecture

End-to-end software support:
- Programming interface
- Compiler optimization

Concise yet powerful Instruction Set Architecture (ISA)

Lightweight programmable architecture for image processing domain
Key Contributions

- **iPIM**: A decoupled control-execution architecture
Key Contributions

• Lightweight programmable arch: **A decoupled control-execution architecture**
• Flexible ISA support: **Single-Instruction-Multiple-Bank (SIMB) ISA**
Key Contributions

- Lightweight programmable arch: A decoupled control-execution architecture
- Flexible ISA support: Single-Instruction-Multiple-Bank (SIMB) ISA
- End-to-end compilation flow: Halide-iPIM
iPIM: High-level Arch Design Overview

• 3D-stacking, near-bank processing-in-memory architecture
• Hierarchical design with good scalability
  • Cube – Vault – Process Group (PG) – Process Engine (PE)
  • A Process Engine (PE) contains a DRAM bank and simple logic components

More details on the next page
Vault Architecture

• Key idea: **Decoupled Control-Execution Architecture**
  • Front-end (complex logic) **control components** of the core are placed on the base logic die;
  • Back-end (simple logic and memory-intensive) **execution components** are placed on the DRAM dies.
Vault Architecture

• Key idea: **Decoupled Control-Execution Architecture**
  • Front-end (complex logic) control components of the core are placed on the base logic die;
  • Back-end (simple logic and memory-intensive) execution components are placed on the DRAM dies.
Vault Architecture

- Key idea: **Decoupled Control-Execution Architecture**
  - Front-end (complex logic) control components of the core are placed on the base logic die;
  - Back-end (simple logic and memory-intensive) execution components are placed on the DRAM dies.
Vault Architecture

- **Key idea:** Decoupled Control-Execution Architecture
  - Front-end (complex logic) control components of the core are placed on the base logic die;
  - Back-end (simple logic and memory-intensive) execution components are placed on the DRAM dies.
Vault Architecture

- Key idea: **Decoupled Control-Execution Architecture**
  - Front-end (complex logic) control components of the core are placed on the base logic die;
  - Back-end (simple logic and memory-intensive) execution components are placed on the DRAM dies.
Vault Architecture

- **Key idea: Decoupled Control-Execution Architecture**
  - Front-end (complex logic) control components of the core are placed on the base logic die;
  - Back-end (simple logic and memory-intensive) execution components are placed on the DRAM dies.
Single-Instruction-Multiple-Bank (SIMB) ISA

- Enable massive bank-level concurrent execution to exploit data parallelism

**Example:** load data from the DRAM bank to the local data register file (DataRF)

*Massive bank-level concurrent execution*

Uniform PE accesses

Instruction broadcast

---

ld_rf

Arbiter

SIMB Controller

Active bank
Single-Instruction-Multiple-Bank (SIMB) ISA

- Enable massive bank-level concurrent execution to exploit data parallelism
- SIMD interface to exploit abundant bank-level bandwidth

Example: load data from the DRAM bank to the local data register file (DataRF)
Single-Instruction-Multiple-Bank (SIMB) ISA

- Predicate execution (*simb_mask*) to allow divergent bank-level accesses / computations

![Diagram](image)
Single-Instruction-Multiple-Bank (SIMB) ISA

- Support indirect addressing in image processing domain

**Example:** access pixel\[x_i, \ offset+y_i\]

Perform:
\[\text{dram\_address} = x_i + (\text{offset}+y_i)\times \text{img\_width}\]
Single-Instruction-Multiple-Bank (SIMB) ISA

- Support indirect addressing in image processing domain

**Example:** access pixel\([x_i, \text{offset}+y_i]\)

This address comes from AddrRF

Access local DRAM bank using \texttt{dram\_address} stored in AddrRF
Single-Instruction-Multiple-Bank (SIMB) ISA

• SIMB ISA also supports:
  • Data-dependent calculation
    • mov_drf / mov_arf
  • Remote data access from different vaults / cubes
    • rd_vsm / wr_vsm / req
  • Control flow instructions
    • jump / cjump / calc_crf / seti_crf
  • Synchronization mechanism
    • Sync

• Please refer to the paper for more details
End-to-end compilation support: **Halide-iPIM**

- **Halide**
  - A domain specific programming language and toolchain for image processing
  - It decouples the *algorithm descriptions* and the *schedules to hardware mapping*

---

**Example**: image blur

```plaintext
// Algorithm
Func blurx(x, y) = (in(x - 1, y) + in(x, y) + in(x + 1, y)) / 3.0f;
Func out(x, y) = (blurx(x, y - 1) + blurx(x, y) + blurx(x, y + 1)) / 3.0f;

// Schedule for iPIM
out.compute_root()
  .ipim_tile(x, y, xi, yi, 8, 8)
  .load_pgsm(xi, yi)
  .vectorize(xi, 4);
```
End-to-end compilation support: **Halide-iPIM**

- **Halide**
  - An domain specific programming language and toolchain for image processing
  - It decouples the algorithm descriptions and the schedules to hardware mapping

- **Halide-iPIM**
  - We extend Halide frontend to support customized schedules for iPIM
  - We leverage existing Halide schedules for pipeline fusion and vectorization on iPIM
End-to-end compilation support: Halide-iPIM

- **Halide**
  - An domain specific programming language and toolchain for image processing
  - It decouples the algorithm descriptions and the schedules to hardware mapping

- **Halide-iPIM**
  - We extend Halide frontend to support customized schedules for iPIM
  - We leverage existing Halide schedules for pipeline fusion and vectorization on iPIM
  - We develop three backend optimizations for iPIM

![Halide-iPIM Diagram](image-url)
Halide-iPIM: Frontend Schedules

- Two new schedule primitives:
  - `ipim_tile()`
    - distribute data into different banks
  
  ```
  ipim_tile(x, y, xi, yi, 8, 8)
  ```

Example: image blur

```c
// Algorithm
Func blur(x, y) = (in(x - 1, y) + in(x, y) + in(x + 1, y)) / 3.0f;
Func out(x, y) = (blur(x, y - 1) + blur(x, y) + blur(x, y + 1)) / 3.0f;

// Schedule for iPIM
out.compute_root().
  ipim_tile(x, y, xi, yi, 8, 8).
  load_pgsm(xi, yi).
  vectorize(xi, 4);
```
Halide-iPIM: Frontend Schedules

• Two new schedule primitives:
  • `load_pgsm()`
  • Utilize the scratchpad of a processing group (PG)

Example: image blur

```
// Algorithm
Func blurx(x, y) = (in(x - 1, y) + in(x, y) + in(x + 1, y)) / 3.0f;
Func out(x, y) = (blurx(x, y - 1) + blurx(x, y) + blurx(x, y + 1)) / 3.0f;
```

```
// Schedule for iPIM
out.compute_root().ipim_tile(x, y, xi, yi, 8, 8)
   .load_pgsm(xi, yi)
   .vectorize(xi, 4);
```

```
Currently in PGSM  Load for next stage

PE3  PE2  PE3  PE2  PE3  PE2
PE1  PE0  PE1  PE0  PE1  PE0
PE3  PE2  PE3  PE2  PE3  PE2
PE1  PE0  PE1  PE0  PE1  PE0
```

- Non-overlapping
- Overlapping (Halo)
- Working Set (Current Stage)
- Working Set (Next Stage)
Halide-iPIM: Frontend Schedules

- Leverage existing schedule primitives:
  - `compute_root()`
    - Specify pipeline fusing
  - `vectorize()`
    - Align data to improve utilization of SIMD units

Example: image blur

```cpp
// Algorithm
Func blurx(x, y) = (in(x - 1, y) + in(x, y) + in(x + 1, y)) / 3.0f;
Func out(x, y) = (blurx(x, y - 1) + blurx(x, y) + blurx(x, y + 1)) / 3.0f;

// Schedule for iPIM
out.compute_root().ipim_tile(x, y, xi, yi, 8, 8).
load_pgsm(xi, yi).vectorize(xi, 4);
```
Halide-iPIM: Backend Optimizations

• Optimization objectives:
  - Instruction-level parallelism
  - DRAM row buffer locality

• Our techniques:
  - Register max spanning
  - Instruction reordering
  - Memory order enforcement
Halide-iPIM: Backend Optimizations

Example: Image Brightening

Input image in DRAM | Pixels in register file | Output image in DRAM

ld_rf  comp  st_rf

iPIM instructions
Halide-iPIM: Backend Optimizations

Example: **Image Brightening**

- **Register dependency**
  - `ld_rf (D0x0, R0x0)`
  - `comp (R0x0, R0x0, 0x2, MUL)`
  - `st_rf (D0x40, R0x0)`
  - `ld_rf (D0x1, R0x0)`
  - `comp (R0x0, R0x0, 0x2, MUL)`
  - `st_rf (D0x41, R0x0)`
  - `ld_rf (D0x2, R0x0)`
  - `comp (R0x0, R0x0, 0x2, MUL)`
  - `st_rf (D0x42, R0x0)`
  - ...

- **Register max spanning**
  - `ld_rf (D0x0, R0x0)`
  - `comp (R0x0, R0x0, 0x2, MUL)`
  - `st_rf (D0x40, R0x0)`
  - `ld_rf (D0x1, R0x0)`
  - `comp (R0x1, R0x1, 0x2, MUL)`
  - `st_rf (D0x41, R0x1)`
  - `ld_rf (D0x2, R0x2)`
  - `comp (R0x2, R0x2, 0x2, MUL)`
  - `st_rf (D0x42, R0x2)`
  - ...

- **Eliminate register dependency**
Halide-iPIM: Backend Optimizations

Example: Image Brightening

Load latency stall

\[
\text{ld}_\text{rf} (D0x0, R0x0) \\
\text{comp} (R0x0, R0x0, \text{0x2, MUL}) \\
\text{st}_\text{rf} (D0x40, R0x0) \\
\text{ld}_\text{rf} (D0x1, R0x1) \\
\text{comp} (R0x1, R0x1, \text{0x2, MUL}) \\
\text{st}_\text{rf} (D0x42, R0x2) \\
\text{ld}_\text{rf} (D0x0, R0x2) \\
\text{comp} (R0x2, R0x2, \text{0x2, MUL}) \\
\text{st}_\text{rf} (D0x42, R0x2) \\
\vdots
\]

Overlapping memory access latency

Instruction reordering

\[
\text{ld}_\text{rf} (D0x0, R0x0) \\
\text{ld}_\text{rf} (D0x1, R0x1) \\
\text{comp} (R0x0, R0x0, \text{0x2, MUL}) \\
\text{comp} (R0x1, R0x1, \text{0x2, MUL}) \\
\text{st}_\text{rf} (D0x40, R0x0) \\
\text{st}_\text{rf} (D0x41, R0x1) \\
\text{ld}_\text{rf} (D0x2, R0x2) \\
\text{comp} (R0x2, R0x2, \text{0x2, MUL}) \\
\text{st}_\text{rf} (D0x42, R0x2) \\
\vdots
\]
Halide-iPIM: Backend Optimizations

Example: *Image Brightening*

Memory order enforcement

- All loads to the same row buffer
- Enforce memory order
- All stores to the same row buffer
Evaluations
Area Analysis

Area overhead of added components per DRAM die: 10.71%
  • Conservatively assume 2x area overhead in DRAM process

Area of control logic on base die: 0.92mm² (fits in 3.5mm² extra area per vault)
iPIM (Near-bank Arch) v.s. GPU

Compared to GPU baseline, iPIM achieves:
- 11.02x average speedup
- 79.49% average energy saving
iPIM (Near-bank Arch) v.s. Process-on-base-die

Compared to process-on-base-die solution, iPIM achieves:

• 3.61x average speedup
• 56.71% average energy saving
Effectiveness of iPIM Compiler Optimizations

All three compiler backend optimizations together provide 3.19x speedup compared to unoptimized program

Instruction reordering is most effective: maximize instruction level parallelism

- **opt**: Apply all 3 optimizations
- **baseline1**: No optimizations
- **baseline2**: No register allocation optimization
- **baseline3**: No instruction reordering
- **baseline4**: No memory ordering enforcement
iPIM Key Takeaways:

• Lightweight programmable arch: **A decoupled control-execution architecture**

• Flexible ISA support: **Single-Instruction-Multiple-Bank (SIMB) ISA**

• End-to-end compilation flow: **Halide-iPIM**

• Evaluation results:
  • 11.02x speedup and 79.49% energy savings over state-of-the-art GPU accelerator
  • 3.61x speedup and 56.71% energy savings over the process-on-base-die solution
  • Overall compiler optimizations provide 3.19x speedup over unoptimized baseline
iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture

Thank you!

Q&A