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

Peng Gu†1, Xinfeng Xie†1, Yufei Ding2, Guoyang Chen3, Weifeng Zhang3, Dimin Niu4, Yuan Xie1,4
1Department of Electrical and Computer Engineering, UCSB, Santa Barbara, USA
2Department of Computer Science, UCSB, Santa Barbara, USA
3Alibaba Cloud Infrastructure, Sunnyvale, USA
4Alibaba DAMO Academy, Sunnyvale, USA

Email:{peng_gu,xinfeng,yufeding,yuanxie}@ucsb.edu
Email:{g.chen,weifeng,z,dimin.niu}@alibaba-inc.com

Abstract—Image processing is becoming an increasingly important domain for many applications on workstations and the data center that require accelerators for high performance and energy efficiency. GPU, which is the state-of-the-art accelerator for image processing, suffers from the memory bandwidth bottleneck. To tackle this bottleneck, near-bank architecture provides a promising solution due to its enormous bank-internal bandwidth and low-energy memory access. However, previous work lacks hardware programmability, while image processing workloads contain numerous heterogeneous pipeline stages with diverse computation and memory access patterns. Enabling programmable near-bank architecture with low hardware overhead remains challenging.

This work proposes iPIM, the first programmable in-memory image processing accelerator using near-bank architecture. We first design a decoupled control-execution architecture to provide lightweight programmability support. Second, we propose the SIMB (Single-Instruction-Multiple-Bank) ISA to enable flexible control flow and data access. Third, we present an end-to-end compilation flow based on Halide that supports a wide range of image processing applications and maps them to our SIMB ISA. We further develop iPIM-aware compiler optimizations, including register allocation, instruction reordering, and memory order enforcement to improve performance. We evaluate a set of representative image processing applications on iPIM and demonstrate that on average iPIM obtains 11.02× acceleration and 79.49% energy saving over an NVIDIA Tesla V100 GPU. Further analysis shows that our compiler optimizations contribute 3.19× speedup over the unoptimized baseline.

Keywords—Process-in-memory, Image Processing, Accelerator

I. INTRODUCTION

With the advance of imaging devices and computational photography, image processing is becoming an increasingly important domain on workstations [48] and the data-center [72] platforms for various applications, such as machine learning [12], biomedical engineering [22], and geographic information systems [34]. These image processing workloads usually involve a large amount of data-intensive computations [55], thus motivating the design of domain-specific accelerators for both high performance and energy efficiency.

As a state-of-the-art accelerator for image processing, GPU has achieved great success [28]. However, the memory-wall [63] impedes its further performance improvement as a result of both the characteristics of image processing workloads and the limited bandwidth provided by the compute-centric architecture. On the one hand, image processing applications require high memory bandwidth, since most of their pipeline stages have low arithmetic intensity. Even worse, its heterogeneous pipeline stages can hardly be fused [18], [19], [30], [53], [59], [61] due to the introduction of redundant computations and the degradation of parallelism. On the other hand, the scaling of memory bandwidth provided by the compute-centric architecture is hindered by both the limited number of off-chip I/O pins [63] and costly data movement energy [31]. To validate this bottleneck, we conduct a detailed profiling of representative image processing benchmarks on an NVIDIA Tesla V100 GPU [2] (Sec.III), which shows apparent bandwidth-bound performance bottleneck (57.55% memory utilization vs. 3.43% ALU utilization).

To overcome the bottleneck of memory bandwidth, the 3D-stacking process-in-memory (3D-PIM) architecture provides a promising solution. This architecture embraces higher memory bandwidth by integrating compute-logic nearer to memory. The first kind of 3D-PIM designs, process-on-base-die solution [5], [29], [39], places compute-logic on the base logic die to utilize the cube-internal Through-Silicon-Via (TSV) bandwidth, and demonstrates bandwidth advantages over GPU. To further unleash the bank-level bandwidth of 3D-PIM, the near-bank solution is proposed [3], [64], [73], which closely integrates compute-logic to each bank in the DRAM dies. It can provide around 10× peak bandwidth improvement compared with the previous solution since compute-logic directly accesses the local bank without going through limited TSVs. Therefore, near-bank architecture emerges as a competitive solution to solve the memory bandwidth challenge for the current compute-centric image processing accelerators.

Although near-bank architecture has great potential for accelerating image processing applications, there are still several challenges. First, heterogeneous image processing pipelines exhibit various computation and memory patterns, thus re-
We design a standalone programmable accelerator, iPIM, and propose SIMB (Single-Instruction-Multiple-Bank) ISA for the proposed near-bank accelerator. The SIMB ISA supports SIMD computation which utilizes the bank’s high I/O width (128b), flexible data movement within the near-bank memory hierarchy, control flow instructions that enable index calculation, and synchronization primitives for communication. Third, we develop an end-to-end compilation flow for new Halide schedules for iPIM. This compilation flow extends the frontend of Halide for supporting these new schedules and includes a backend with optimizations for iPIM including register allocation, instruction reordering, and memory order enforcement to reduce resource conflict, exploit instruction-level parallelism, and optimize DRAM row-buffer locality, respectively.

The contributions of our work are summarized as follows:

- We design a standalone programmable accelerator, iPIM, using 3D-stacking near-bank architecture for image processing applications. By using a decoupled control-execution architecture, iPIM supports programmability with small area overhead per DRAM die (∼ 10.71%).
- We propose SIMB (Single-Instruction-Multiple-Bank) ISA which enables flexible computation, data access, and communication patterns to support various pipeline stages in image processing applications.
- We develop an end-to-end compilation flow based on Halide with novel iPIM schedules and various iPIM backend optimizations including register allocation, instruction reordering, and memory-order enforcement.
- Evaluation results of representative image processing benchmarks, including single stage and heterogeneous multi-stage pipelines, show that iPIM design together with backend optimizations can achieve 11.02× speedup and 79.49% energy saving on average over an NVIDIA Tesla V100 GPU. The backend optimizations improve 3.19× performance compared with the naïve baseline.

II. BACKGROUND


We introduce the overall architecture and opportunities of 3D-PIM for image processing applications as follows. One 3D stacking memory cube (HBM [66] and HMC [58]) consists of multiple stacked DRAM dies on top of a base logic die. The control logic on the base logic die will access memory using TSVs (Through-Silicon-Vias) [71], which are vertical interconnects shared among 3D layers. Previous work [5], [29], [39] has explored placing compute-logic on the base logic die to harvest cube-internal TSV bandwidth. This solution is constrained by the maximal bandwidth provided by TSVs (currently 307GB/s for one cube with 1024 TSVs [66]), and scaling TSVs is very difficult due to the large area overhead (already 18.8% of each 3D layer [66]). To tackle this TSV bottleneck, researchers further explore near-bank architecture for 3D-PIM [3], [64], [73]. The near-bank design integrates simple compute-logic adjacent to each bank without changing DRAM bank circuitry. The enormous bank-level bandwidth and massive bank-level parallelism make it promising to accelerate data-intensive image processing applications. However, the lack of programmability due to expensive control core support is very challenging to enable heterogeneous image processing pipelines which have diverse computation and memory patterns. In this work, we tackle this programmability challenge by proposing a lightweight decoupled control-execution architecture (Sec.IV-B), and SIMB ISA that supports a wide range of image processing pipelines (Sec.IV-C).

B. Image Processing and Halide Programming Language

Image processing contains heterogeneous pipelines which are wide and deep [61], and it is bound by memory bandwidth on the compute-centric architecture. From the applications’ point of view, first, most image processing pipeline stages have low arithmetic intensity (operations per byte) and massive data parallelism for individual pixels, such as elementwise and stencil computations. Second, the whole pipelines are long and heterogeneous (e.g., 23 different stages in local Laplacian filter [56]), and they have complex data dependencies (e.g., resampling and gather in local Laplacian filter). Therefore, from the hardware’s point of view, these features make it very difficult to apply pipeline fusion techniques [18], [19], [30], [53], [59], [61] to boost the performance. Thus, on compute-centric accelerators like GPU, image processing performance is bound by memory-bandwidth (Sec.III), and near-bank architecture provides a promising solution.

Although widely-adopted programming languages for image processing like Halide [61] provide optimizations for a wide range of compute-centric accelerators like GPU [52] and FPGA [21], there are no existing solutions for memory-centric
accelerators. Halide decouples the algorithm descriptions and 
the algorithm to hardware mapping, thus programmers can 
separately describe an algorithm and a schedule. Based on 
the provided algorithms and schedules, the Halide compiler 
will synthesize hardware-specific programs. In this work, we 
propose the first end-to-end compilation framework for image 
processing applications in Halide on the near-bank architecture 
by designing novel schedules (Sec.V-B) and developing a 
compiler backend to improve the performance (Sec.V-C).

III. MOTIVATION

First, we find that memory-bandwidth is the performance 
bottleneck for GPU, which is the current state-of-the-art image 
processing accelerator [28]. We conduct a detailed profiling 
of representative benchmarks (Table.II) using Halide framework [61] and DIV8K [65] dataset on an NVIDIA Tesla V100 
GPU [36]. The measured total DRAM bandwidth, DRAM 
utilization, and ALU (both FP32 and INT32) utilization are 
shown in Fig.1(a). We observe that these benchmarks ex-
hibit DRAM bandwidth-bound behavior by achieving 57.55% 
DRAM utilization (518GB/s bandwidth) and 3.43% ALU 
utilization on average. We also note that the memory and 
ALU utilization are both low for Histogram benchmark, which 
results from that Histogram involves value-dependent computa-
tions and the Halide schedule for GPU cannot achieve ideal 
performance.

Second, we observe that multi-stage benchmarks (the last 4 
in Fig.1), which are optimized by Halide pipeline fusion, show 
early performance improvement compared with single-stage 
benchmarks (the first 6 in Fig.1). The ALU utilization only 
increases from 2.85% to 4.53%. Also, the DRAM utilization is 
merely reduced from 58.80% to 55.73%, which is still 
significantly higher than the ALU utilization. We conclude 
that Halide compiler optimizations cannot change the memory-
bound behavior of image processing applications on GPU, 
motivating an accelerator providing more memory bandwidth.

Third, we find that index calculation, which is an important 
part of programmability support for flexible memory access 
patterns, consumes a large portion of total ALU utilization 
for image processing workloads. For the current profiling, 
index calculation uses INT32 data type and algorithm-related 
computation uses FP32 data type. The breakdown of the ALU 
utilization is shown in Fig.1(b). We observe that on average 
index calculation takes 58.71% of total ALU utilization, and 
index calculation dominates the total ALU utilization (> 60%) 
for 5 out of 10 benchmarks. The index calculation ratio is high 
because image processing requires frequent translations from 
2D image to 1D memory space [20]. This motivates us to 
enable architecture support for index calculation in iPIM.

IV. ARCHITECTURE DESIGN

First, we introduce the microarchitecture overview in 
Sec.IV-A. Second, we describe iPIM’s decoupled control-
execution scheme in Sec. IV-B. Then, we explain the instruc-
tion set architecture design in Sec. IV-C. Next, we discuss the 
remote memory access mechanism and present the method 
for inter-vault synchronization in Sec.IV-D. In the end, we 
detail the functionalities of iPIM’s hardware components in 
Sec.IV-E.

A. Microarchitecture Overview

In general, iPIM uses the 3D-stacking near-bank archi-
tecture with a top-down hierarchy of cube, vault, process 
group, and process engine as illustrated in Fig.2(a). First, 
iPIM consists of multiple cubes (Fig.2(a1)) interconnected by 
SERDES links similar to HMC [58]. Second, one cube is 
horizontally partitioned into multiple vaults (usually 16 per 
cube) connected by an on-chip network. Each vault (Fig.2(a2)) 
spans multiple 3D-stacking layers, including several process-
in-memory (PIM) dies (usually 4 to 8 per vault) and one 
base logic die. The inter-layer communication is realized by 
Through-Silicon-Vias (TSVs, usually 64 per vault), which are 
high-bandwidth vertical interconnects that link each layer to 
the base logic die. The base logic die of each vault contains 
one iPIM control core (Fig.2(b)), which is the basic unit to 
execute an iPIM program. Next, one PIM die of each vault 
contains one process group (PG) (Fig.2(a3)), which further 
consists of many process engines and a shared process group 
scratchpad memory (PGSM). Last but not least, each process 
engine (PE) (Fig.2(c)) employs near-bank architecture, where 
compute-logic and lightweight buffers are integrated with a 
DRAM bank. Especially, each PE adds an address register file 
and an integer ALU to efficiently support index calculations 
which are important for image processing (Fig.1(b)).

Based on this microarchitecture, iPIM decouples the control, 
which happens on the base logic die, from the massive bank-
level parallel execution, which happens on the PIM dies 
(Sec.IV-B). In addition, we design SIMB ISA to support various 
computation and memory access patterns in image pro-
cessing, and efficiently move data among the iPIM hierarchy 
(PE-level, PG-level, vault-level, or cube-level) (Sec.IV-C).

B. Decoupled Control-Execution Architecture

iPIM uses a novel decoupled control-execution design to 
reduce the overhead of the control core by placing it on the
to benefit from the abundant bank-

Depending on the program counter (pc), an instruction will be fetched from the instruction cache ($I$) and decoded. pc can be updated from control register file (CtrlRF) using jump/cjump, and calc_crf, seti_crf are used to calculate control flow values. The decoded instruction will be checked against instructions in the Issued Inst Queue. If true/anti/output data dependency is found, the instruction will stall with a pipeline bubble inserted. Once the instruction is issued, it is added to the Issued Inst Queue until retirement. The issued instruction is broadcast by SIMB controller to each PE according to the simb_mask, or sent to a vault-level unit for execution (e.g. seti_vsm). If the instruction involves remote vault access, it is dispatched to the network interface controller (NIC). (a) For the vault-local SIMB execution, each PE will check the corresponding bit in simb_mask and proceed execution or stay idle. (b) For the remote vault access, the request will be translated into packets and traverse the on-chip network or off-chip links. The SIMB instruction executes in lock-step, and an instruction retires only if all bits in the simb_mask are cleared. Each time a PE finishes an instruction, the SIMB controller will clear its execution bit. After an instruction finishes, it is committed by popping the corresponding entry from the Issued Inst Queue. This also clears data dependency for later instructions.

As a conclusion, this architecture not only enables lightweight programmability to control heterogeneous pipeline stages (base logic die) but also supports parallel execution to provide abundant memory bandwidth for data-intensive image processing operations (PIM dies).
high-level overview, this ISA resembles a RISC-like SIMD ISA that enables bank-parallel computation as well as efficient memory access as detailed below.

For the computation, we highlight the support for SIMB and SIMD execution. To enable SIMB, each SIMB-capable instruction has a simb_mask field, which is a boolean vector indicating whether the corresponding PE should execute this instruction or not. For example, in a vault with 8 PGs where each PG has 4 PEs, the simb_mask should be a 32b boolean vector. To enable SIMD, each computation and data movement instruction operates on a vector of FP32/INT32 elements. The vector length is chosen to be 4 to match the local bank’s interface (128b per access) and TSV’s data transfer width (128b per cycle), so the internal bandwidth is fully utilized. For each vault, control signals and data signals share the same physical TSVs through time multiplexing, which is realized by the arbiter in Fig.2. Therefore, there is no additional TSV area cost for control signals to each PE.

For memory access, we emphasize the support for data movement and memory indexing. To enable data movement, SIMB ISA contains different instructions to realize customized data flow along the memory hierarchy. To support flexible indexing, SIMB ISA contains index calculation instructions and allows communication between the address register file and the data register file to enable data-dependent computation.

More detailed explanations about SIMB ISA are as follows:

The computation instruction (comp) supports vector-vector/(-scalar) operations specified by the mode field. The vec_mask indicates which positions in the vector are valid for computation. The op defines the operation to be performed.

The index calculation instruction (calc_arf) supports parallel address calculations among PEs, so each PE can have independent memory access patterns. To allow different PEs inside a vault to operate on different addresses in the SIMB fashion, indirect addressing is supported for the bank (dram_addr), PGSM (pgsm_addr), and VSM (vsm_addr) addresses. When indirect address mode is used, the corresponding address field will first index into the address register file in each PE, and then the fetched address will be used to index the target memory component. This can satisfy the need for flexible 2D memory access patterns in image processing.

The data movement instructions (intra-vault/inter-vault) are classified into two types. The first type involves DRAM bank access (st/ld_rf, st/ld_pgsn for local vault access, and req for remote vault access). The second type includes data movement along the memory hierarchy within a vault.

The control flow instructions support control flow (jumps/cjumps) and related calculations (calc_cr_f, seti_cr_f). These enable iPIM programs to have dynamic behaviors to support various computation patterns in image processing.

The synchronization instruction (sync) allows different vaults to synchronize computation stages according to a phase_id. Sec.IV-D contains a detailed example.

D. Remote Access and Synchronization

iPIM supports data access from a remote vault by implementing an asynchronous request instruction (req). First, the local vault needs to provide the remote vault’s memory address and issues a req to local vault’s NIC. Then, the remote vault adds this request to the DRAM request queue of the corresponding PE. Next, the accessed data is temporarily buffered in the remote vault’s VSM and sent back to the local vault after inter-vault link traversal. The communication interface guarantees delivery, so no acknowledgment is required.

iPIM realizes synchronization among different vaults through a lock-step synchronization instruction (sync), which acts as a barrier to block all instructions after this sync. The synchronization relies on a centralized master-slave protocol, where a selected vault is designated as the master vault and all other slave vaults are coordinated. For a vault, a synchronization point is reached only if all instructions before that sync finish execution. Then, the slave vault will signal the master vault, after which the master vault will update a global synchronization status vector. After the global synchronization point is reached, the master vault will broadcast a proceed_phase message to all slave vaults, and all vaults will commit the sync instruction and proceed execution phases.

E. Hardware Components and Usages

This section introduces important information regarding the hardware components in iPIM as follows:

Data/Address Register File (DataRF/AddRF): Both the DataRF and AddRF employ multi-port architecture to avoid resource hazards during execution. The DataRF has a vector interface (128b) that aligns with the bank’s width. To accommodate the scalar interface (32b) of AddRF, a multiplexer is added. In addition, AddRF locations A0-A3 are reserved to store PE’s peID, pgID, vaultID, and chipID, respectively.

Process Group Scratchpad Memory (PGSM): PGSM is used for data sharing among PEs in a PG. To access another PE’s memory, a simple way is to generate a ld_pgsn from the source PE followed by a rd_pgsn to the destination PE with the same PGSM address. To enable parallel PE access, PGSM allocates individual ports for each PE and employs multi-bank architecture. (Fig.2(a3)). Each PE has a separate read port and a write port into PGSM so that data loading to PGSM can be overlapped with PGSM access. Also, PGSM has a 2D memory abstraction for image processing applications.

Vault Scratchpad Memory (VSM): VSM has three functionalities. First, VSM is used for data sharing among PEs in a vault. To access another PG’s memory, a possible solution is to generate a ld_rf and a wr_vsm to write the data to a VSM location, and use a rd_vsm to bring the data to local PE. Note that TSVs are shared among PGs, so VSM has only one data port for TSVs. Second, VSM temporarily buffers the data for remote vault access. Third, VSM acts as the instruction memory that accepts computation offloading from a host.

In-DRAM Memory Controller: iPIM integrates a lightweight memory controller that serves the banks inside each PG (Fig.2(c)). The memory controller contains a memory request
// 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);

Listing 1. Code example of image blur.

queue, a DRAM command buffer, DRAM command translation and issuing logic, a counter to record last DRAM command issuing cycle, a DRAM status register, and an open row address register. Currently the memory controller supports two page policies (open/close page) and two DRAM scheduling policies (FCFS,FR-FCFS) [62]. It also schedules DRAM refresh commands according to tREFI and tRFC timing parameters similar to AxRAM [73].

On/off-chip Network: iPIM adopts a 2D mesh topology for both the on-chip and off-chip network. Each router assumes Input-Queued (IQ) microarchitecture and implements the X – Y routing algorithm. Also, simple flow control and channel allocation policies [37] are used.

V. COMPILER

This section details the design of an end-to-end compilation flow based on Halide for iPIM hardware. First, we introduce the programming interface of iPIM in Sec.V-A, which includes the design of new schedules for iPIM. Second, we explain the compilation flow in Sec.V-B including the extension of Halide front-end compilation passes and our customized backend for iPIM. Finally, we detail the backend optimizations in Sec.V-C for generating efficient iPIM executable programs.

A. Programming Interface

To support various image processing applications composed of heterogeneous pipelines on iPIM, we use Halide as the programming language because of its success in this application domain. Our front-end support for Halide eases the burden of programmers from two perspectives. First, the image processing algorithm written in Halide does not have to be changed for iPIM because Halide decouples the algorithm from its scheduling. Second, we develop customized schedules to provide an easy-to-use high-level abstraction for indicating workload partition and data sharing among PEs in iPIM. Thus the workload partition and data sharing are optimized automatically by our end-to-end compilation flow according to these high-level schedules without programmers’ involvement.

We develop customized schedule primitives to efficiently exploit hardware characteristics on iPIM hardware. In particular, we extend Halide with two new schedule primitives, ipim_tile() and load_pgsm(), for distributing data into different banks and utilizing the scratchpad of a processing-group. The first customized schedule for iPIM, ipim_tile(), specifies the dimensions of image data to be partitioned and distributed across the hierarchy of iPIM. For example, the schedule ipim_tile(x, y, xi, yi, 8, 8) in Listing 1 indicates that the image will be partitioned into image tiles (8x8 size). In addition to the partition of the image into tiles, this schedule also indicates the distribution of these image tiles across all PEs. Fig.3(a) shows the distribution of these image tiles into different levels in the hierarchy of iPIM. Specifically, image tiles are distributed in an interleaved way to the PEs of the same PG so that they can load adjacent image tiles at the same loop iteration to improve data sharing. The second customized schedule for iPIM, load_pgsm(), indicates the usage of shared scratchpad memory at the PG level. For example, the schedule load_pgsm(xi, yi) in Listing 1 indicates that the data of input image needed for computing output along loops xi and yi will be loaded into shared scratchpad memory before using it for the computation. Fig.3(b) shows the usage of PGSM according to the specification of load_pgsm() in Listing 1 at PG-level. After loading data into PGSM, Fig. 3(c) shows the temporal scheduling of the computation for each PE including four steps (2) to load the whole region of input data (1) for a vector of output pixels. By supporting this schedule, data sharing across adjacent image tiles can happen at the PG level.

In addition to our customized schedules for data partition and sharing on iPIM, we leverage existing Halide schedules to specify the fusion of pipelines and the vectorization of computation on iPIM. In Listing 1, compute_root() ensures that the loops along dimensions of the Func out will be outermost loops and the stages of computing blurx will be fused into the computation of out. During code generation, each compute_root() implies a kernel function reading input data from and writing output results to DRAM banks. Besides compute_root(), we also exploit the vectorization schedule (vectorize(xi, 4)) supported by Halide for iPIM because our
ISA includes SIMD instructions. Specifically, we exploit the compilation pass of vectorization in Halide frontend aligning data to improve the utilization of SIMD units in iPIM.

B. Compilation Flow

As shown in Fig. 4, we develop an end-to-end compilation flow to support an automatic transformation from a Halide algorithm with customized iPIM schedules to a hardware executable program on iPIM. We develop the frontend code transformation to support our iPIM schedules and the backend instruction optimizations to improve the performance of generated programs. Our backend optimizations have unique challenges due to our novel near-bank architecture from two perspectives. First, because of the simple in-order control core design, our register allocation phase needs to prevent data hazards due to register contention. Thus, this phase aims to span virtual registers into different physical registers to avoid such data hazards instead of minimizing the number of allocated registers in the typical register allocation phase. Second, our instruction reordering phase needs to optimize row buffer locality when exploiting the instruction-level parallelism (ILP) because of the timing characteristics of DRAM banks. Thus we add new virtual dependencies to enhance the row-buffer locality which is critical to the performance of programs. In summary, our end-to-end compilation flow takes advantage of customized schedules to generate programs exploiting iPIM hardware features, such as PGSM, and our backend optimizations further improve the performance of the programs.

C. Backend Optimization

In this section, we detail the novel instruction optimizations we developed for the backend. The major goal of the backend in our compilation flow is to generate efficient iPIM executable programs from the input Halide module. The backend decouples this program generation process into two parts, instruction lowering which translates the Halide module into iPIM instructions, and instruction optimizations which improve the performance of the generated programs. We will detail these optimizations into three parts, register allocation, instruction reordering, and memory order enforcement. The effectiveness of our backend optimizations will be quantitatively analyzed in Sec.VII-E.

Register Allocation: The goal of register allocation is to assign a physical register to each virtual register and avoid the instruction dependency due to the conflict of physical registers. To avoid such conflicts on physical registers, our algorithm is based on the depth-first search on the register interference graph, and it attempts to assign each virtual register from a physical register different from the most recently used one. The input of our algorithm, the register interference graph, is built upon the traditional liveness analysis of virtual registers. After building the register interference graph and converting the register allocation problem into a graph coloring problem, our algorithm tries to avoid the conflict of physical registers rather than solely minimizing the number of physical registers in the allocation. Because the architecture design of iPIM uses simple in-order control core to avoid hardware overheads, the traditional register allocation method could cause the dependency between instructions due to the conflict of physical registers, which further leads to pipeline stalls.

Instruction Reordering: Although the program generated by register allocation is already executable on iPIM, we reorder instructions in the program to maximally exploit the instruction-level parallelism. Because of the instruction issue mechanism of our in-order core, the dependency between adjacent instructions will lead to pipeline stalls. Therefore, the instruction reordering aims to expose instruction-level parallelism to the hardware, which eliminates pipeline stalls and improves the performance. We first build a directed graph where each node stands for instruction and directed edges between nodes represent the dependency between instructions. Then we develop our instruction reordering algorithm which traverses this directed graph in topological order. We associate each node with a timestamp to provide an estimation of its earliest time ready to be issued \(T(v)\) in Algorithm.1. When there are multiple instructions available at a time step, we will schedule the load instruction with the \(T\) smaller than the current time step or the node with the smallest \(T\). After marking the instruction to be scheduled, we update the incoming degree of all its outgoing neighbors, and also their timestamp \(T\). Finally, all instructions are scheduled into the output sequence after iterating through \(|V|\) time steps. This graph traversal algorithm is demonstrated in Algorithm 1, and its time complexity is \(O(|V|\log|V| + |E|)\) where \(|V|\) is the number of nodes, i.e. instructions, and \(|E|\) is the number of edges in the directed dependency graph.

Memory order enforcement: In addition to data depen-
Fig. 5. Instruction reordering example: image brighten.

In Sec.VII-D, we study the instruction breakdown of each benchmark. In Sec.VII-E, we show the benefits of iPIM’s compiler optimizations by conducting a series of comparative evaluations. In the end, we conclude that iPIM’s compiler optimizations are near-optimal by showing the achieved high hardware utilization and instruction per cycle (IPC) number.

A. Experimental Setup

Benchmark and Dataset Selection. As detailed in Table.II, we use a set of single-stage and multi-stage benchmarks for an in-depth and comprehensive analysis. The single-stage benchmarks cover a wide range of computation and memory patterns in important image processing operations [13], such as elementwise, stencil, reduction, gather, shift, and other data-dependent operations. With them, we are able to provide isolated in-depth analysis for each image processing operation. The multi-stage benchmarks, which are widely used in image processing programming languages [19], [30], [41], [61], on the other hand, contain heterogeneous pipeline stages that require the support of programmability. We use DIV8K [65] dataset, which contains over 1500 images covering diverse scene contexts with 8K (7680 \times 4320) resolution for all the evaluated benchmarks. The choice of a high-resolution dataset is to reflect the application trend on workstations and data-center that deep learning training, medical image processing, and geographical information system require higher image quality.

Hardware Configuration. iPIM assumes 3D-stacking memory configuration similar to previous near-bank accelerators [3], [64], [73] without changing DRAM’s core timing. We list the detailed hardware configuration, latency values, energy consumption, and DRAM settings in Table.III. We also consider important timing parameters to limit power ($t_{RRDS}=4$, $t_{RRDL}=6$, $t_{FAW}=16$). iPIM contains 8 iPIM cubes (total $\sim 850\text{mm}^2$) to compare with a Tesla V100 GPU card [2] with 4 HBM stacks (total $\sim 1199\text{mm}^2$), where one HBM stack consumes $\sim 96\text{mm}^2$ footprint [66].

Simulation Methodology. We develop a cycle-accurate simulator extended from ramulator [40] by integrating cus-
TABLE III

<table>
<thead>
<tr>
<th>Parameter Names</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cubes/Vaults/PES/PEs/InstQueue/DRAMReqQueue</td>
<td>8/128b/4</td>
</tr>
<tr>
<td>SIMD_len / CAS_width / link_width (SERDES)</td>
<td>4/128b/4</td>
</tr>
<tr>
<td>Bank / AddrRF / DataRF / PGSIM / VSM (Byte)</td>
<td>16M/256/1K/8K/256K</td>
</tr>
<tr>
<td>I/CX / BCD / CCD / RD / WR / IPM (ns)</td>
<td>1/1/4/24/4/4/4</td>
</tr>
<tr>
<td>tADDRRF / tDataRF / tPGSM / tVSM (ns)</td>
<td>1/1/1/1</td>
</tr>
<tr>
<td>tADDRSUB / tMULT / tMAC / tLOGIC (ns)</td>
<td>4/5/0/1</td>
</tr>
<tr>
<td>tPEbus / tTSV / tNoC (hop) / tSERDES (hop) (ns)</td>
<td>1/1/0/08</td>
</tr>
<tr>
<td>RD,WR / PREACT / AddrRF / DataRF (access)</td>
<td>0.5tmem/0.23tact/0.43/tdata</td>
</tr>
<tr>
<td>tADD(SUB) / tMUL / tMAC / tLOGIC (ns)</td>
<td>4.5/0/1</td>
</tr>
<tr>
<td>tADDRRF / tDATA RF</td>
<td>0.017/0.004/0.006</td>
</tr>
<tr>
<td>PEbus / tTSV / tSERDES (hit)</td>
<td>0.017/0.004/0.006</td>
</tr>
<tr>
<td>DRAM_rowbuffer_policy / DRAM_schedule</td>
<td>open_page / FR-FCFS</td>
</tr>
</tbody>
</table>

Fig. 6. Throughput and speedup comparison between iPIM and GPU.

TABLE IV

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Area (mm$^2$)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD Unit</td>
<td>64</td>
<td>2.26</td>
<td>2.36</td>
</tr>
<tr>
<td>Int ALU</td>
<td>64</td>
<td>0.32</td>
<td>0.33</td>
</tr>
<tr>
<td>Address Register File</td>
<td>64</td>
<td>0.20</td>
<td>0.21</td>
</tr>
<tr>
<td>Data Register File</td>
<td>64</td>
<td>1.79</td>
<td>1.86</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>16</td>
<td>1.84</td>
<td>1.92</td>
</tr>
<tr>
<td>PGSM</td>
<td>16</td>
<td>3.87</td>
<td>4.03</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>10.28</td>
<td>10.71</td>
</tr>
</tbody>
</table>

B. Performance, Energy-efficiency, Area, and Thermal Issues

Performance. iPIM achieves 11.02× average speedup over the GPU as shown in Fig.6. From the hardware’s point of view, this speedup is mainly attributed to iPIM’s ample memory bandwidth as a result of near-bank architecture (more comparisons in Sec.VII-C). From the software’s point of view, this high speedup is achieved through good compiler optimizations (more analysis in Sec.VII-E).

Next, we explain the variations in the speedup for different benchmarks. First, the Brighten benchmark consists of elementwise operations which are completely bound by memory bandwidth, so iPIM’s enormous bank-level bandwidth can provide very good speedup (21.09×). Second, the Histogram benchmark involves data-dependent computation resulting in inferior performance using Halide’s default schedule on GPU. The schedule on iPIM converts it into a reduction of parallel reduced partial histogram results, thus it achieves significant performance improvement (43.78×). Third, Blur andStencil Chain benchmarks only have moderate speedup (4.32× and 4.30×, respectively) on iPIM. Later analysis (Sec.VII-D and Sec.VII-E2) shows that these two benchmarks have higher computation intensity than other benchmarks, and involve a lot of index calculations which are bound by address register file. As a conclusion, the results indicate that iPIM can effectively accelerate a wide range of image processing applications.

Energy-efficiency. iPIM achieves 79.49% average energy saving over the GPU (Fig.7). The energy saving mainly comes from the reduction of expensive data movement compared with GPU, since iPIM’s compute-logic can use the local bank without off-chip data access. Sec.VII-C2 provides a more detailed energy breakdown to show the small overhead of data movement in iPIM. Also, we observe that for each benchmark the energy saving in Fig.7 is approximately proportional to the speedup in Fig.6. This is because iPIM’s increased bank-level bandwidth is a result of near-bank data access, which also contributes to the reduction of data movement energy.

Next, we explain the difference in energy saving between single-stage benchmarks and multi-stage benchmarks (89.26% and 66.81%, respectively). iPIM employs compute_root schedule, where intermediate data between pipelines are written back to banks without fusing. In comparison, since Halide employs pipeline fusion for multi-stage benchmarks on GPU, the expensive off-chip memory access can be reduced due to increased-on-chip data reuse. As a result, iPIM has a slight drop in energy saving for multi-stage benchmarks.

Area. iPIM’s decoupled control-execution architecture is area-efficient because it only adds small area overhead (execution part) per DRAM die and the control core can be well fitted on the base logic die. First, we evaluate the area of execution components in the PIM layers considering DRAM process overhead (Fig.2(c)), and normalize the total added area to a DRAM die (96nm$^2$ [66]). We show that the added area per DRAM die is small (10.71%) to support programmability according to Table.IV. Second, we evaluate the area of iPIM’s control core on the base logic die (Fig.2(b)). The core consumes 0.92mm$^2$ total silicon footprint (including
the VSM which takes 0.23mm²), and it can be well fitted into the extra area of each vault (3.5mm² [29]) on the base logic die. On the contrary, if this control core is naively integrated with each bank, the total area overhead per DRAM die will increase to 122.36%, which is 10.42 × larger than that of our decoupled control-execution design.

**Thermal Issues.** iPIM’s peak power is 63W per cube considering both DRAM dies and the base logic die, and the peak power density is 593mW/mm². The normal operating temperature for 8Hi HBM2 DRAM dies is 105°C [66], and we conservatively assume the DRAM dies in our case operates under 85°C. A prior study on 3D PIM thermal analysis [75] shows that active cooling solutions can effectively satisfy this thermal constraint (85°C). Both commodity-server active cooling solution [51] (peak power density allowed: 706mW/mm²) and high-end-server active cooling solution [25] (peak power density allowed: 1214mW/mm²) can be used. Also, compared with previous work [75] where PIM logics are concentrated on the base logic die far from the top heat sink, iPIM distributes the PIM logics evenly to each DRAM dies, so the heat dissipation will be much better [4]. In addition, we note that the majority of the peak power (78.5%) is induced by simultaneously activating/precharging DRAM banks. Since iPIM compiler optimizes row buffer locality for image processing workloads, for memory-intensive workloads with ideal row buffer locality, the frequency of this activity is relatively low.

**C. Architecture Analysis**

1) **Comparison of iPIM and process-on-base-die solution:** We compare iPIM with the process-on-base-die (PonB) solution and observe that iPIM on average achieves 3.61 × speedup and 56.71% energy saving as shown in Fig. 8. We further explain the PonB configuration and the advantages of iPIM over the PonB solution. The only difference of PonB with iPIM is that all near-bank components are moved to the base logic die, and these components access their DRAM banks through TSVs. We evaluate PonB using the same benchmarks and simulator while serializing the data traffic on the shared TSVs between the base logic die and the DRAM dies. The inferior performance of the PonB solution is because all memory accesses need to go through TSVs with limited bandwidth, which is only 10% of iPIM’s peak memory bandwidth. The energy overhead of the PonB solution is induced by expensive in-cube data movement energy, which is 2.48 × of iPIM’s local bank access energy. We argue that it is impractical for the PonB solution to have the same memory bandwidth as iPIM by increasing the number of TSVs, since this will increase the TSV overhead by 10 ×, which translated to 187% area overhead per DRAM die.

2) **Energy Breakdown:** We provide a detailed energy breakdown of iPIM programs shown in Fig.9. The DRAM in this figure contains the background energy, activation/precharge (RAS) energy, read/write (CAS) energy, and refresh energy. The SIMD unit contains all floating/integer operation energy of the SIMD unit. The AddrRF/DataRF/PGSM contains the read/write energy and leakage energy. The Others contains data movement energy and control core’s energy on the base logic die. The breakdown shows that iPIM’s decoupled execution-control architecture spends most of the energy on PIM dies (89.17%), and only a small part on data movements and the control core (10.83%). This can be further justified by the instruction breakdown analysis in Sec.VII-D. The low energy consumption of inter-vault and intra-vault data movement is contributed from (1) iPIM’s near-bank architecture and (2) localized data movement benefited from the memory hierarchy and compiler optimizations.

3) **Sensitivity Analysis:** We conduct sensitivity studies on how the number of registers per PE (RF) and Process Group Scratchpad size (PGSM) will impact the execution time. First, to study the RF sensitivity (Fig.10(a)), we vary the RF value from 16 to 128 and normalize the execution time to the case when RF=128. We observe that RF=16, RF=32, and RF=64 have 46.8%, 26.8%, and 9.5% performance drop compared to RF=128, respectively. The performance drop is attributed to the decreased number of registers that results in (1) more

![Fig. 7. Energy comparison between iPIM and GPU.](image-url)

![Fig. 8. Comparison of near-bank and process-on-base-die solutions.](image-url)

![Fig. 9. Energy breakdown of iPIM programs.](image-url)
registers spilling to the local DRAM bank, and (2) increased register data dependency. Second, to study the PGSM sensitivity (Fig. 10(b)), we change the PGSM from $2\text{KB}$ to $8\text{KB}$ and normalize the execution time to the case when PGSM=$8\text{KB}$. We observe that PGSM=$2\text{KB}$ and RF=$4\text{KB}$ have $58.9\%$ and $39.0\%$ performance drop compared to PGSM=$8\text{KB}$, respectively. This is because reduced scratchpad size will increase the number of accesses to long latency DRAM. For this paper, we choose RF=$64$ and PGSM=$8\text{KB}$ as a tradeoff between performance and area overhead (Table.IV).

D. Instruction Breakdown

From the instruction breakdown of iPIM programs shown in Fig.11, we can observe that each benchmark is a combination of different instructions with varied ratios. From the programmability perspective, this indicates that SIMB ISA efficiently maps heterogeneous pipeline stages which exhibit diverse computation, data movement, and control flow patterns. Therefore, SIMB ISA can provide flexible support for a wide range of image processing applications.

We also find that index calculation instructions on average take $23.25\%$ of total instruction count. Since the image uses 2D memory abstraction and physical memory assumes linear address space, frequent index calculations are required to map the 2D image reference locations to the corresponding memory addresses. This index calculation overhead takes more than $28\%$ of total instruction count for Blur, Shift, Histogram, Bilateral Grid, and Stencil Chain benchmarks. This provides a direct explanation about iPIM’s moderate speedup on these benchmarks as shown in Fig.6 except for Histogram.

Another important observation is that the inter-vault data movement instructions are a very small part ($1.44\%$) of the total instruction count. This confirms image processing kernels have good data parallelism and can be efficiently mapped onto near-bank architecture with little global data movement.

E. Compiler Analysis

1) Effectiveness of compiler optimizations: We add a set of comparative evaluations to justify the performance benefits provided by iPIM’s compiler optimizations (Fig.12). We summarize these optimization choices as follows. The register allocation policy determines whether to use the minimum number of physical registers ($\text{min}$) or scatter registers to avoid the dependency of instructions ($\text{max}$). The instruction reordering option determines whether to reorder the instruction of programs generated by the register allocation stage. The memory order enforcement option chooses whether to add dependency edges on adjacent memory requests or not before sending the dependency graph to the instruction reordering stage. The optimized design (opt) adopts the $\text{max}$ register allocation policy and applies both instruction reordering and memory order enforcement. The naïve baseline (baseline1) assumes the $\text{min}$ register allocation policy without instruction reordering. All of baseline2, baseline3, and baseline4 have only one different compiler optimization option compared to opt. Specifically, baseline2 uses the $\text{min}$ register allocation policy, baseline3 does not apply instruction reordering, and baseline4 does not enforce memory order. The rest of settings for baseline2 – 4 remain the same as opt.

We observe that all of iPIM compiler optimizations provide an overall $3.19\times$ speedup (opt v.s. baseline1). Further analysis shows that $\text{max}$ register allocation provides $2.59\times$ speedup than $\text{min}$ register allocation (opt v.s. baseline2). This is because iPIM’s in-order core does not support expensive register renaming mechanism in the out-of-order execution, and $\text{max}$ register allocation can optimally eliminate output-dependency and anti-dependency to prevent issue stall of later instructions. Next, instruction reordering provides $2.74\times$ speedup (opt v.s. baseline3), since it can expose more instruction-level-
parallelism by overlapping instructions without dependency. In the end, enforcement of memory-order provides 1.30× speedup (opt vs. baseline). The reason is that it can maximally interleave other instructions with memory access requests and it also improves row buffer locality.

2) IPC and Utilization analysis: As shown in Fig. 13, we provide the IPC of the control cores and the utilization of key hardware components on the PIM dies. First, we observe that the average IPC achieves a very high value of 0.63 after intensive compiler optimizations. This implies that iPIM currently attains near-optimal performance, and further improvement has an upper bound of 1.59× assuming no pipeline stalls. Second, detailed analysis shows that for each benchmark, key hardware components reach very high utilization. For example, benchmarks with intensive index calculations (Blur, Shift, Histogram, Bilateral Grid, and Stencil Chain) realize more than 40% utilization on the address register file. As a conclusion, the high IPC and hardware utilization indicate current compilation flow has well-optimized image processing applications on iPIM architecture.

VIII. RELATED WORK

Image processing accelerators. Previous work has explored Field Programmable Gate Array (FPGA) [19], [41], [59], Coarse Grain Reconfigurable Arrays (CGRA) [67], [68], and ASIC solutions [20], [49] for image processing acceleration. These accelerators target mobile and embedded platforms, where power-efficiency and low latency are the primary goals of optimization. Also, the application scenarios are mostly image streaming applications with a small working set, so spatially distributed data flow architecture is often adopted to map the entire image processing pipeline. To achieve the desired power-efficiency, line buffer [67] is widely used to exploit producer-consumer data locality and fuse pipeline stages, so the intermediate data can stay on-chip without expensive off-chip memory access. In comparison, iPIM focuses on data center and workstation environments, where the complex algorithm pipelines and large working set due to high-resolution images need both high memory capacity and bandwidth. Thus, conventional compute-centric accelerators will suffer from the memory bandwidth bottleneck, while iPIM’s near-bank architecture provides abundant bandwidth resources to tackle this challenge.

Process-in-memory (PIM) accelerators. We compare iPIM with previous work using practical DRAM technology without invasive modifications to the DRAM bank’s circuitry [46], [47]. The first category of research [26], [38], [57] that tries to integrate processor cores to DRAM dies suffers from large area overhead. iPIM solves this challenge by proposing a control-execution decoupled approach where the control core is placed on the base logic die and shared by execution units closely integrated with each bank. The second category of research adopts 3D process-on-logic-die architecture [5], [10], [29], [33], [39], [45], [54], [70], [74], [76], which is bound by the TSV bandwidth available in the base logic die. iPIM evaluation shows 3.61× speedup and 56.71% energy saving compared with this approach. To further improve memory bandwidth, a few work [3], [64], [73] employs near-bank architecture similar to this work, but only supports limited fixed functionalities and cannot map the heterogeneous image processing pipelines which have diverse computation and memory patterns. iPIM proposes SIMB ISA and an end-to-end compilation flow to solve this programmability challenge. In addition, some recent work proposes integrating computation logic on the DRAM DIMM modules to enable low overhead near-data processing [6], [7], [32], [43]. While practical, these architecture designs only have small bandwidth improvement over CPUs compared with 3D-PIM solutions. There are also studies exploiting PIM architectures based on non-volatile memory (NVM) technologies [8], [17], [27], [35], [44], [69]. Compared with these studies, DRAM provides a better write endurance than NVM, which is critical to image processing applications where intermediate results of pipelines need to be written back to memory.

IX. CONCLUSION

This paper proposes iPIM, the first programmable in-memory image processing accelerator using near-bank architecture. iPIM uses a decoupled control-execution architecture to support lightweight programmability. It also contains a novel SIMB (Single-Instruction-Multiple-Bank) ISA to enable various computation and memory patterns for heterogeneous image processing pipelines. In addition, this paper develops an end-to-end compilation flow extended from Halide with new schedules for iPIM. The compiler backend further contains optimizations for iPIM including register allocation, instruction reordering, and memory order enforcement. Evaluations show that iPIM supports programmability with small area overhead, and provides significant speedup and energy saving compared with GPU. Further analysis demonstrates the benefits of iPIM compared with the previous process-on-base-logic architecture design and the effectiveness of iPIM’s compiler optimizations.

REFERENCES


