 MPU: Towards Bandwidth-abundant SIMT Processor via Near-bank Computing

Xinfeng Xie†1, Peng Gu†1, Yufei Ding2, Dimin Niu3, Hongzhong Zheng3, Yuan Xie1,3

1Department of Electrical and Computer Engineering, UCSB, Santa Barbara, USA
2Department of Computer Science, UCSB, Santa Barbara, USA
3Alibaba DAMO Academy, Sunnyvale, USA

Email:{xinfeng, peng_gu, yufei_ding, yuan_xie}@ucsb.edu
Email:{dimin.niu, hongzhong.zheng}@alibaba-inc.com

Abstract—With the growing number of data-intensive workloads, GPU, which is the state-of-the-art single-instruction-multiple-thread (SIMT) processor, is hindered by the memory bandwidth wall. To alleviate this bottleneck, previously proposed 3D-stacking near-bank computing accelerators benefit from abundant bank-internal bandwidth by bringing computations closer to the DRAM banks. However, these accelerators are specialized for certain application domains with simple architecture data paths and customized software mapping schemes. For general purpose scenarios, lightweight hardware designs for diverse data paths, architectural supports for the SIMT programming model, and end-to-end software optimizations remain challenging.

To address these issues, we propose MPU (Memory-centric Processing Unit), the first SIMT processor based on 3D-stacking near-bank computing architecture. First, to realize diverse data paths with small overheads while leveraging bank-level bandwidth, MPU adopts a hybrid pipeline with the capability of offloading instructions to near-bank compute-logic. Second, we explore two architectural supports for the SIMT programming model, including a near-bank shared memory design and a multiple activated row-buffers enhancement. Third, we present an end-to-end compilation flow for MPU to support CUDA programs. To fully utilize MPU’s hybrid pipeline, we develop a backend optimization for the instruction offloading procedure. The evaluation results of MPU demonstrate 3.46× speedup and 2.57× energy reduction compared with an NVIDIA Tesla V100 GPU on a set of representative data-intensive workloads.

I. INTRODUCTION

Nowadays, parallel computing platforms play an increasingly important role in accelerating emerging data-intensive applications from various domains such as deep learning [42], image processing [23], and bioinformatics [25]. In particular, a general purpose graphics processing unit (GPGPU) with its single-instruction-multiple-thread (SIMT) programming model benefits these workloads by exploiting massive memory-level parallelism and providing DRAM bandwidth higher than traditional CPUs. Despite the success of GPGPU in accelerating these workloads, its further performance scaling is constrained by the “memory bandwidth wall” [65] challenge. From the application’s perspective, workloads from these data-intensive applications usually require a large number of memory accesses with little computation. This characteristic makes memory bandwidth the dominating factor of performance. Unfortunately, from the technology’s point of view, the scaling of main memory bandwidth for the compute-centric architecture is confined by both the insufficient number of off-chip I/O pins [40] and the expensive data movement energy [24]. To further corroborate this bandwidth-bound behavior, in Sec.II we evaluate a set of representative benchmarks on an NVIDIA Tesla V100 GPU [1], and find that on average the off-chip memory bandwidth utilization (55.90%) overwhelmingly surpasses the ALU utilization (2.57%) as shown in Fig. 1.

The 3D-stacking near-data-processing (3D-NPD) architecture [72] emerges as a promising approach to alleviate this memory bandwidth bottleneck. Currently, high-end GPUs [43], [59] are equipped with high-bandwidth memory (HBM) stacks [68], where off-chip data transfers need to go through the low performance I/Os on the silicon interposer. The principal idea of 3D-NPD is to closely integrate affordable logic components adjacent to the memory stack. A large number of pioneering studies have adopted the processing-on-base-logic-die (PonB) architecture, where general purpose cores (e.g., SIMT cores [25], [51], [52], [49], [61], [71], [72]) are placed on the base logic die of the 3D stack to benefit from the intra-stack bandwidth enhancement (around 2× w.r.t. HBM [72]). This solution provides a mediocre bandwidth improvement because intra-stack memory accesses are still bounded by the limited number of through-silicon vias (TSVs) between memory dies and the base logic die. To overcome this bandwidth bottleneck of TSVs, recent near-bank accelerators [43], [23], [67], [76] further move simple arithmetic units closer to the DRAM banks to harvest the abundant bank-internal bandwidth (around 10× w.r.t. processing-on-logic-die solution [23]). These near-bank accelerators have demonstrated significant speedups (around 2× −14× w.r.t. GPU) thus they are promising to tackle the GPU’s memory bandwidth issue.

Despite its potential to provide plentiful memory bandwidth, there are still several challenges for near-bank computing in accelerating general purpose data-intensive workloads. First, the pipeline of SIMT processors contains complex logic components (e.g., load-store-unit [56] and large register files [39]). Different from the prior near-bank accelerators customized

* Xinfeng Xie and Peng Gu are co-primary authors.
for applications, the SIMT pipeline is needed for general purpose data-intensive workloads. Naively placing the whole pipeline with complex logic components and complicated data paths in the DRAM die introduces an intolerable area overhead [18], [56], [60]. Second, the efficient support of the SIMT programming model on near-bank computing is needed, especially the inter-thread communication and the dynamic scheduling of warps [35]. As the shared memory is frequently used for inter-thread communication in a thread block, directly placing it on the base logic die will incur enormous TSV traffic. The dynamic scheduling of warps could disrupt the row-buffer locality of DRAM banks, seriously downgrading bandwidth utilization. Third, it requires both the end-to-end support of a parallel programming language to ease the programmers’ burden and backend compiler optimizations for near-bank computing to exploit hardware potentials.

To address these challenges, we design MPU (Memory-centric Processing Unit), the first SIMT processor based on 3D-stacking near-bank computing architecture, and an end-to-end compiler flow supporting CUDA programs [54] with optimizations tailored to MPU. First, we design a hybrid SIMT pipeline for MPU where only a small number of registers and other lightweight components are added on the DRAM dies. At runtime, instructions are fetched, decoded, and issued on the base logic die while they can be offloaded to near-bank units (NBU) according to either compiler hints or hardware policies. To facilitate this hybrid execution of instructions, we propose an instruction offload engine to make instruction movement decisions, a register track table and a register move engine to flexibly transfer registers, and a load-store unit extension to handle near-bank load/store requests. Second, we propose two architectural optimizations for the SIMT model. For the shared memory, we move it to the DRAM die and restructure the core organization by placing all NBUs associated with the same core on the same DRAM die. For the dynamic scheduling of thread warps, we enable multiple activated row-buffers per DRAM bank to reduce the ping-pong effect thus improving the bandwidth. Third, it requires both the end-to-end compilation flow to support CUDA programs.

To optimize instruction offloading location on MPU’s hybrid pipeline, we further propose a novel instruction and register location annotation algorithm through the static analysis of instructions, which effectively reduces the data movement among the shared TSVs.

The contributions of our work are summarized as follows:

- To the best of our knowledge, we design the first near-bank SIMT processor using a hybrid pipeline with an instruction offloading mechanism. By integrating lightweight hardware components on the DRAM die, MPU achieves a small area overhead for general purpose processing.

- We propose two architectural optimizations for the SIMT model, including the near-bank shared memory to reduce data movement and multiple activated row-buffers to alleviate ping-pong effects in the dynamic warp scheduling.

- We develop an end-to-end compilation flow supporting

CUDA programs on MPU and a novel backend optimization annotating the locations of registers and instructions.

- Evaluation results of representative data-intensive workloads show that MPU with all optimizations achieves 3.46× speedup and 2.57× energy reduction on average over an NVIDIA Tesla V100 GPU.

II. MOTIVATION

Despite the success of the graphics processing unit (GPU) in accelerating data-intensive parallel programs, we observe from performance characterizations that the memory bandwidth is a serious performance challenge for these workloads on the state-of-the-art GPU. Specifically, we evaluate a set of representative data-intensive workloads from various application domains including deep learning, bioinformatics, linear algebra, and image processing applications as detailed in Table I. The measured memory bandwidth, bandwidth utilization, and compute utilization of an NVIDIA Tesla V100 GPU [44] are shown in Fig. 1. On average, these benchmarks achieve 55.90% DRAM bandwidth utilization (503.10 GB/s) and 2.57% compute utilization. The saturation of DRAM bandwidth and the low utilization of the compute resources exhibit a memory-bandwidth bound behavior. This performance characteristic results from the low arithmetic density and the regular memory access patterns in most of these workloads. Also, we note that the workloads HIST and NW show relatively low bandwidth utilization as a result of the long memory access latency on GPU.

For workloads suffering from either the limited DRAM bandwidth or the long DRAM access latency on GPU, near-bank computing is a promising architecture to alleviate these performance bottlenecks because of both abundant bank-level memory bandwidth and reduced memory access latency. However, prior near-bank computing accelerators [8], [23], [67], [76] are domain-customized, since they have simple data paths, application-specific mapping strategies, and inefficient general purpose programming language support. The lack of programmability for these accelerators confines them to a niche application market, adding non-recurring engineering costs in manufacturing. Moreover, parallel data-intensive workloads usually come from various application domains, making none of these near-bank accelerators feasible to support all of these parallel programs.

In summary, the memory bandwidth bottleneck on the state-of-the-art GPU urges the need for a higher memory bandwidth for data-intensive parallel programs, and the huge overheads
of placing an SIMT processor near banks introduce unique technical challenges. Both of these factors motivate us to design MPU, the first general purpose SIMT processor based on 3D-stacking near-bank computing to exploit bank-level bandwidth and alleviate the GPU bandwidth bottleneck.

III. BACKGROUND

3D-stacking Near-data Computing (3D-NDC): 3D-NDC is based on commercially available 3D-stacking memory modules (HBM [68] and HMC [37]) with DRAM dies stacked on the top of a base logic die. Previous GPU-style processing-on-logic-die solutions [25], [31], [32], [49], [61], [71], [72], [77] usually place simple SIMT cores (Sec.III) on the base logic die, and the cores access memory using TSVs (Through-Silicon-Vias) [75], which are vertical interconnects shared among 3D layers. However, this solution is limited by the available bandwidth provided by TSVs (currently 307 GB/s for one stack with 1024 TSVs [68]), and scaling TSVs is very difficult due to the large area overhead (already 18.8% of each 3D layer [68]). To solve the TSV challenge, recent work pushes simple compute logic adjacent to each bank to utilize the enormous bank-level bandwidth for domain-specific acceleration [3], [23], [67], [76]. However, it is challenging to support general purpose near-bank architecture due to the large area overhead of fabricating general purpose cores in DRAM dies. We solve this challenge by a novel hybrid architecture that decouples the SIMT pipeline and duplicates some parts of the backend stages onto the near-bank DRAM dies (Sec.IV-B). The SIMT model also has some special features that require optimizations. First, the shared memory is extensively used for inter-thread communication inside the same thread block. Placing it in the base logic die may introduce extra communication traffic among the 3D layers. Second, the SIMT scheduling causes warps to access different row-buffers interchangeably, causing a row-buffer ping-pong effect. MPU explores architectural optimizations, near-bank shared memory and multiple activated row-buffers, for these two features detailed in Sec.IV-C.

IV. ARCHITECTURE DESIGN

First, we discuss MPU’s high-level design in Sec.IV-A. Second, we introduce MPU’s hybrid pipeline in Sec.IV-B and further describe its instruction offloading mechanism in Sec.IV-B1 and hybrid load-store unit in Sec.IV-B2. Next, we present two architectural optimizations for SIMT model in Sec.IV-C.

A. Microarchitecture Overview

From the high-level, MPU adopts a scalable design with many processors (Fig.2 1) interconnected by an off-chip network (similar to SERDES links in the HMC [37]) as shown in the bottom of Fig.2 1. Each processor is a 3D-stacking cube of a base logic die stacked with multiple DRAM dies, connected by vertically shared buses called the through silicon vias (TSVs) [75] (Fig.2 5). The base logic die is horizontally partitioned into an array of SIMT cores (Fig.2 3) interconnected by an on-chip 2D-mesh network [29].

To harvest the near-bank bandwidth with small overheads on DRAM dies, MPU’s SIMT core adopts a hybrid pipeline design. In the MPU core (Fig.2 2), complex logics are placed on the base logic die, and some lightweight components in
the execution stage are replicated on near-bank locations. On the base logic die, a core consists of four subcores (Fig. 2), an instruction cache, a warp scheduler, and components for handling inter-core traffic (network interface unit, router, and LSU-Remote). The TSVs (Fig.2) are evenly divided among the cores (64b data buses per core), via which the subcores can communicate with near-bank components on DRAM dies. All the core’s near-bank components are located within the same DRAM die, containing four near-bank units (NBUs) (Fig.2) and the shared memory. To enable efficient processing in this hybrid architecture (Sec.IV-B), we propose a novel instruction offloading mechanism and a hybrid load-store unit design.

In addition, MPU considers two architectural optimizations for the SIMT programming model in Sec.IV-C. First, we find that naively implementing shared memory on the base logic die results in poor performance, so we restructure the core’s 3D organization and develop a near-bank shared memory design. Second, we observe that the dynamic execution of SIMT warps may disrupt the row-buffer locality of DRAM banks, so we adopt a technique to enable multiple activated row-buffers inside the same DRAM bank.

### B. Hybrid Pipeline

As illustrated in Fig.2 (2), an MPU core (Fig.2 (2) 2) adopts a hybrid pipeline design that is split between the base logic die (subcore) (Fig.2 (2) 3) and the DRAM die (near-bank unit, NBU) (Fig.2 (2) 4). The frontend components of the SIMT pipeline mostly comprise of control flow and data dependency logic, so they are mainly contained in the subcores, including instruction fetch (I-cache, SIMT stack, warp table), decode, and issue (scoreboard) stages. For the backend pipeline, MPU duplicates some simple parts from the subcore to the NBU, including the register file, operand collectors, and ALUs. Other complex units such as LSU and network interface units are left on the base logic die. Note that the memory controller and the shared memory are entirely moved from the base logic die to the DRAM die, since near-bank memory controller will reduce TSV traffic for DRAM commands [24, 69], and shared memory can reduce TSV traffic for register movement (Sec.IV-C).

In addition to the original pipeline components, to assist flexible instruction offloading, each subcore also adds an instruction offload engine, a register move engine, and an associated register track table (Sec.IV-B1). Besides, the load-store unit (LSU) is modified and augmented to support remote data traffic and near-bank instruction offloading (sec.IV-B2).

#### 1) Instruction Offloading Mechanism:

The instruction offload engine after the issue stage will decide whether to offload the instruction for near-bank execution (Near-bank instr. data path in Fig.3). The offloaded instruction will first travel through the TSVs, then access the near-bank operand collector to collect operand data from the near-bank register file. Then, the arithmetic and logic computation instructions will be sent to the near-bank ALUs, and the ld/st instructions will be provided to the LSU-Extension for further processing, where the shared memory or the DRAM controller is involved. After the execution finishes, the resulting registers will be written back into the near-bank register file, and the instruction is returned to the subcore for the final commit, where the scoreboard clears its data dependency.

The first step (Fig.3 1) is to identify the target instruction’s location according to three policies with decreasing priority. The first policy will set the instruction with the far-bank location if the corresponding operation type (Opcode) falls in the far-bank operation set. For example, since address range checking and memory coalescing can only be performed by the LSU in the subcore, currently ld/st.global instructions are classified as far-bank locations. If the first hardware policy cannot determine the location, then the compiler hint associated with the instruction will determine whether this instruction will be offloaded to NBU or not. If the instruction has no compiler hints, then the hardware will check the register track table. The instruction will be offloaded to NBU if all source registers have valid near-bank copies. Otherwise, the instruction will be passed to the far-bank. This default policy takes the far-bank subcore as a fall-back location for all instructions as it has the full pipeline support.

The second step (Fig.3 2) is to determine the locations for the source registers using the hardware policy or the instruction location derived in the first step. For ld/st.global, the hardware policy will set the address register location to be far-bank, since it is required by the LSU. The data register location is set to near-bank. For ld/st.shared, both the address register and the data register are set to near-bank location. If the hardware policy is not given, all the source and destination registers’ locations will follow the instruction’s location.

The third step (Fig.3 3) will move registers to their corresponding locations if they are not currently available in the register track table (Register Movement data path in Fig.2 (2)). For example, the instruction offload engine may require register %r1 to be valid in the far-bank register file, while the register track table indicates %r1 only exists in the near-bank register file (e.g., FBValid is False but NBValid is True). Then, the far-bank register move engine initiates a request to the near-bank register move engine, which then reads %r1 from the near-bank register file and returns it to the far-bank register move engine. The far-bank register move engine will write %r1 into the far-bank register file, and the instruction
Fig. 4. Three load-store unit (LSU) components’ microarchitecture using ld.global data path as an example. (1) LSU in each subcore, (2) LSU-Remote in each core, (3) LSU-Extension in each near-bank unit offloading engine will start instruction offloading once all registers are in the target locations. Note that we optimize register locations in Sec.V-B, so a hit in the register track table will not cause register movement. In the end, the register track table is updated to reflect the most current register location information.

2) Hybrid Load-store Unit (LSU):

The original LSU in each subcore is augmented with the LSU-Remote in each core to handle remote traffic (Remote ld/st.global data path in Fig.2 (2)), and the LSU-Extension in each NBU to handle both near-bank instruction offloading and local DRAM transactions (Local ld/st.global data path in Fig.2 (2)). We will introduce them and use ld.global instruction as an example in the following paragraphs.

LSU: As shown in Fig.4 (1), after receiving a ld.global instruction, the LSU first performs address range checking (Fig.4 1) to split the instruction to remote access and local access. If there is remote access, it is encoded and sent to the network interface unit (Fig.4 2) to request remote data. For the local access, the following steps are performed concurrently. First, the LSU will check if all the SIMT mask fields are valid or not to determine thread divergence (Fig.4 3). Second, the LSU will perform memory coalescing (Fig.4 4) on the local memory addresses. It will also judge if all the memory addresses are perfectly coalesced, meaning that the load request will access a continuous DRAM address space. Third, the LSU will compare the NBU_id field of the generated DRAM addresses with the NBU_id associated with the register in the given warp (Fig.4 5). The ld.global instruction is decided for near-bank offloading (Fig.4 6) only if all threads are valid, register and DRAM addresses have the same NBU_id, and all DRAM addresses are coalesced. Note that since all of the above assumptions are satisfied, we only need to transfer the leading DRAM address, register ID, and the NBU_id. The SIMB mask will be ignored and restored in the LSU-Extension side. If the above conditions cannot be met, the LSU will issue DRAM transactions to the LSU-Extension (Fig.4 7) and gather returned local DRAM data. Combined with the returned remote DRAM data, the final DRAM data will be used to compose a register write request and transferred for near-bank writeback. The reason to load the DRAM data first to the near-bank register file is that it can benefit near-bank execution due to the reduction of TSV traffic. For far-bank execution, the register data will eventually be brought down to the far-bank register file, so there is no increase in the TSV traffic.

LSU-Remote: As shown in Fig.4 (2), LSU-Remote receives remote ld.global request and decodes them into a series of DRAM addresses and DRAM NBU_id. It then sends these DRAM transactions to the LSU-Extension through the TSVs. After receiving the returned DRAM data, it encodes it together with the source core location and request ID and composes a response packet, finally sending it back to the original core’s LSU who requests this DRAM data.

LSU-Extension: As shown in Fig.4 (3), LSU-Extension has two data paths. In Fig.4 (3-a), it handles DRAM transaction requests from the TSVs by sending the DRAM addresses to the memory controller, and sends back the returned DRAM data through TSV either to the LSU-Remote or the LSU. In Fig.4 (3-b), it handles offloaded local ld.global instruction by first restoring the full address list from the leading DRAM address. Then, it sends the DRAM addresses to the memory controller, gathers returned DRAM data, and stores the data into the near-bank register file according to the register ID.

C. Optimizations for the SIMT model

Near-bank Shared Memory Design: The shared memory is extensively used for inter-thread communication in the same thread block for a great number of important GPU benchmarks [35]. If the default shared memory location is set in the far-bank subcore on the base logic die, a lot of register data movement traffic will be created and the TSVs will be congested, causing significant performance loss. Thus, it will be desirable to enable a near-bank shared memory design and
set the default register location for ld/st.shared to the near-
bank register file. However, this is impossible in the vertical
core structure (Fig5(1)) in the default HMC-style setting [37],
where all the NBUs associated with a core are distributed
among multiple 3D stacks. Under such an assumption, moving
the shared memory (Snem) to each NBU means that the
shared memory is split into each 3D layer and inter-thread
shared memory accesses need to go through the bandwidth-
bound TSVs. To enable the near-band shared memory, we
restructure the core’s 3D-organization as shown in the hori-
zontal core design in Fig5(2). In our solution, we put all
NBUs of the same core into the same DRAM die, so that
all NBUs can access the near-band shared memory without
TSV’s constraints. In Sec VI-C, we confirm the benefits of
this optimization on benchmarks that extensively use shared
memory.

Multiple Activated Row-Buffers Design: The dynamic
execution of warps will create a ping-pong effect on DRAM’s
row-buffer. Ideally, warps from the same thread block execut-
ing the same memory access instruction will have continuous
DRAM addresses and result in a high row-buffer hit rate. How-
ever, the hardware dynamically issues available instructions
from each warp, resulting in the ping-pong effect of different
warps accessing a few row-buffers irregularly. Since MPU has
no hardware cache, this ping-pong effect will cause frequent
DRAM precharge and activations, significantly downgrade its
performance.

To solve this issue in a software transparent way, we observe
that for a lot of benchmarks we evaluate, only a small set of
row-buffers are active in a short period. If we can enable multi-
ple row-buffers to be simultaneously activated, then this ping-
pong effect will be greatly alleviated. Based on the design of
MASA (Multitude of Activated Subarrays) [33], which enables
multiple subarrays’ row-buffers to be activated in parallel, we
change our address mapping so that continuous DRAM row
addresses will be mapped to interleaved subarrays’ physical
row. Extra row address latches and access transistors are added
to enable different warps to access different row buffers in
independent subarrays. Through evaluations in Sec VI-C, we
confirm that this design can decrease the row-buffer miss rate
and increase performance for multiple benchmarks.

V. PROGRAMMING INTERFACE AND COMPILER

Sec. V-A introduces the role of MPU in a heterogeneous
platform and its programming interface. Sec. V-B details the
compiler support of transformations from high-level CUDA
kernels to optimized programs running on the MPU.

A. Programming Interface

MPU acts as a standalone accelerator in a heterogeneous
platform with a similar usage of the GPU. In terms of the
memory system abstraction, MPU has its own memory space
independent from the host. In order to use MPU as an acceler-
ator, the host is responsible to allocate the device memory on
MPU, transfer input data to MPU, launch computation kernel
// CUDA kernel for scalar-vector multiplication
__global__ void ScalarVectorMultiply(float* inpu,
float* output, float alpha, int len) {
  int numThreads = gridDim.x * blockDim.x;
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  for (int i = tid; i < len; i += numThreads) {
    output[i] = alpha * inpu[i];
  }
}

int main() {
...

// Memory allocation on MPU
mpu_malloc(mp_output_vec, len * sizeof(float));
mpu_malloc(mp_output_vec, len * sizeof(float));
// Transfer the input data to MPU
mpu_memcpy(mp_output_vec, host_input_vec,
  len * sizeof(float), Host2Device);
// Launch kernel for the computation on MPU
ScalarVectorMultiply<<<GridCfg, BlockCfg>>>(
  mp_output_vec, mp_output_vec, alpha, len);
// Transfer MPU computation results
mpu_memcpy(host_output_vec, mp_output_vec,
  len * sizeof(float), Device2Host);
...}

Listing 1. Code example of scalar-vector multiplication.

Fig. 6. The backend of MPU compiler generating MPU executable kernels
from PTX kernels.

to MPU, and transfer computation results from MPU. Listing 1
is a code example of offloading scalar-vector multiplication
to MPU. The main function contains code to handle the memory
allocation (mpu_malloc), memory transfers (mpu_memcpy),
and the kernel launch. These functions are implemented in
MPU runtime for the communication between the host and
MPU. During the kernel launch, MPU runtime is also respon-
sible to dispatch the workload of thread blocks to MPU
cores according to the thread block configurations. To ease
the burden of implementing kernels on MPU, MPU supports
CUDA programming language as a realization of the SIMT
programming model. The __global__ function in line 2 of
Listing 1 for implementing the scalar-vector multiplication
totally follows the CUDA syntax. As a result, we can leverage
a GPU compiler as our front-end to parse the source code
and generate intermediate instructions. Then, our compiler
backend as detailed in Sec. V-B is responsible for backend
optimizations tailored for MPU architecture.

B. MPU Compiler

To enable the SIMT programming model, MPU supports an
end-to-end compilation flow from CUDA programs to MPU
executable programs. This compilation flow contains a novel
static analysis stage to optimize the location assignment of in-
structions by reducing data movement between MPU cores and
near-bank units. Experimental results in Sec VI-D demonstrate
Algorithm 1: Location annotation algorithm

Input: A kernel with a list of instructions $I$
Output: A location $L$ for all the registers and instructions.
$L(reg)$: the location of a register $reg$.
$L(instr)$: the location of an instruction $instr$.

1. Init the location of all registers to $U$.
2. Init the location of all instructions to $U$.
3. Init the set of registers $R = \emptyset$.

// Annotate the initial location to address registers, value registers, and
// control registers.
for instr in $I$ do
    for reg in $InstrSrcRegs \cup instr.DstRegs$ do
        $R = R \cup \{reg\}$
        if instr.type $\in \{ld.global, st.global\}$
            $L(instrSrcRegs) = F$
        if instr.type == ld.global
            $L(instrSrcRegs) = F$
            $L(instrDstRegs) = N$
        if instr.type == st.global
            $L(instrSrcRegs) = N$
            $L(instrDstRegs) = N$
        if instr.type $\in \{ld.shared, st.shared\}$
            $L(instrSrcRegs) = N$
            $L(instrDstRegs) = N$
        // Propagate the location of known registers to others.
        while $vreg \in R, L(vreg) \neq unknown$ do
            for instr in $I$ do
                for reg in $InstrSrcRegs$ do
                    if $L(reg) == U$ then
                        $L(reg) = L(instr.DstRegs)$
                    if $L(reg) == L(instr.DstRegs)$ then
                        $L(reg) = B$
                // Annotate the location of instructions according to the location of their
                // destination registers.
                for instr $\in I$ do
                    $L(instr) = L(instr.DstRegs)$

that this novel location annotation improves the performance of MPU compared with a default hardware policy.

The end-to-end compilation flow of MPU includes frontend stages and backend stages. In frontend stages, the MPU compiler reuses \texttt{nvcc} \cite{55} to compile CUDA programs \cite{55} to generate kernels in Parallel Thread Extension (PTX) \cite{55} ISA which is a kind of intermediate representation of CUDA kernels. Then, the backend generates the MPU hardware executables from the PTX kernels, which includes three main stages as shown in Fig\[\textbf{6}]. Among these three stages, the branch analysis and register allocation stages are common to support the SIMT programming model. The branch analysis stage infers the re-convergence point of each jump instruction so that the hardware can maintain a SIMT stack to handle thread divergence efficiently during the execution [\textsuperscript{21}]. This problem can be formulated as the post-dominator analysis of a control-flow graph representing the program. The register allocation stage analyzes the liveness of each virtual register in the program to build a register interference graph. The allocation of physical registers can then be formulated as a graph coloring problem on this register interference graph.

The location annotation is a novel backend stage to optimize the performance by annotating the location of instructions as either the near-bank NBU or the far-bank subcore on the base logic die. As shown in Fig\[\textbf{5}]

kernels on MPU, the locations annotated on instructions will be used to finalize the runtime instruction offloading decision as explained in Sec\[\textbf{IV-B}]. The main idea of this optimization is a heuristic approach based on the static analysis extracting the dependency chains of the control-related, address-related, and value-related registers. First, value-related registers will be annotated as near-bank registers while control-related and address-related registers will be annotated as far-bank registers. Then, instructions from the dependency chain of value-related registers are annotated as the near-bank instructions while the rest of instructions are annotated as the far-bank instructions. For example, Fig\[\textbf{7}]

VI. EVALUATION

In Sec\[\textbf{VI-A}]

we introduce the experiment setup and methodologies. In Sec\[\textbf{VI-B}]

we show the performance, area, energy, and thermal results of MPU. In Sec\[\textbf{VI-C}]

we demonstrate the benefit of MPU’s architecture optimizations and the comparison with the prior processing-on-base-logic-die designs. In Sec\[\textbf{VI-D}]

we present the effectiveness of the location annotation in our compiler backend optimization.
A. Experimental Setup

**Benchmark.** To evaluate the effectiveness of the MPU design in supporting data-intensive parallel programs, we select a set of representative CUDA workloads as shown in Table I. In particular, these workloads are from various important application domains including image processing, machine learning, linear algebra, and bioinformatics. Because our MPU compiler needs either CUDA source code or PTX kernels to generate MPU executable programs, we have implementations of these workloads from either well-known GPU benchmarks, such as Rodinia [14], or writing CUDA programs in the same functionality while achieving performance comparable to state-of-the-art libraries, such as cuBLAS [52], and the TSV and on/off-chip buses adopt parameters from previous studies [15], [59], [63]. For the ALU, we use the measured results from PTX instructions [8], [9]. For area evaluation, we use design compiler [19] to analyse pre-layout area of the vector ALU and the SIMT core pipeline [31]. We use AxRAM’s area result [38] for the in-dram memory controller, and scale it to 20nm. The area for the shared memory, register file, operand collector, and LSU-Extension are derived from cacti [15].

**Hardware Configuration.** Using the 3D-stacking memory configuration similar to the previous near-bank accelerators [3], [23], [67], [76]. MPU needs no changes to DRAM’s core circuit except the multiple activated row-buffers enhancement [33]. The detailed hardware configuration, latency values, energy consumption, and DRAM settings are presented in Table II. MPU contains 8 processors (total ~ 926mm²) to compare with a Tesla V100 GPU card [1] with 4 HBM stacks (total ~ 1199mm²), where one HBM stack consumes ~ 96mm² footprint [68].

**Simulation Methodology.** We develop an event-driven simulator using SimPy [45], which adapts the simulation framework from GPGPU-Sim [11] for SIMT core model, Ramulator [24] for DRAM model, and Booksim [29] for interconnect network model. MPU is designed to run at a clock frequency of 1GHz under the 20nm technology node. For energy and latency modeling, we first use the design compiler to get the power values for the SIMT core pipeline based on Harmonica project [31]. Then, we use cacti [15] to evaluate the register file, shared memory, and the DRAM bank. Since

---

**TABLE I**

| Workload | App Domain | Reference | Description |
|----------|------------|-----------|-------------|
| BLUR | Image Processing | Halde [64] | 3x3 blur. |
| CONV | Machine Learning | TensorFlow [3] | 3x3 conv. |
| GEMV | Linear Algebra | CUB [51] | Matrix-vector multiply. |
| HIST | Image Processing | TensorFloat [2] | Histogram. |
| KMEANS | Machine Learning | Rodinia [14] | K-means clustering. |
| KNN | Machine Learning | Rodinia [33] | K-nearest-neighbour. |
| TRANS | Linear Algebra | cuBLAS [52] | Tensor transposition. |
| MAXP | Machine Learning | TensorFloat [2] | Max-pooling. |
| NW | Bioinformatics | Rodinia [14] | Sequence alignment. |
| UPSAMP | Image Processing | Halde [62] | Image upsample. |
| AXPY | Linear Algebra | CUB [51] | Vector add. |

**TABLE II**

| Parameter Names | Configuration |
|-----------------|---------------|
| Proc/3D-Core/Subcore,NBU/Bank/RowBuf | 8(4,16)(4,4/4/4) |
| SIMT/Bank/IOVS/on/off/chip_bus (Bit) | 32/256b(1024/256b)128 |
| Bank/Cache/Far/Near-bank REF/Smem (Byte) | 16M/128K/32K/16K/64K |
| rC/D/C/CDM/RT/RPR/GR/As/KFC/REFI [3] | 14/2/4/14/33/150/5900 |
| fCore / fTSV / fRouter / f(on)offchip_bus (GHz) | 1/2/2/2 |
| RD/WP/RE/ACT/REF/R/SMEM [15] (access) | 0.15ps/0.27ns/1.93ns/40.0ps/22.2ps |
| Operand_collector / LSU-Extension (Access) | 41.49ps/39.67ps/350ps/3900ps |
| TSV [59] (on/off-chip bus [15], [63]) (J/bit) | 4.53ps/0.72ps/4.5ps |
| DRAM_rowbuffer_policy / DRAM_schedule | open_page / PR-FCFS |

---

Fig. 8. (1) Execution time and speedup comparison with the GPU. (2) Workloads memory intensity and speedup. The major components in the operand collector and the LSU-Extension are SRAM buffers, we also use cacti to evaluate their latency and energy values. We model the router latency and energy consumption using BookSim2’s model [29], and the TSV and on/off-chip buses adopt parameters from previous studies [15], [59], [63]. For the ALU, we use the measured results from PTX instructions [8], [9]. For area evaluation, we use design compiler [19] to analyse pre-layout area of the vector ALU and the SIMT core pipeline [31]. We use AxRAM’s area result [38] for the in-dram memory controller and scale it to 20nm. The area for the shared memory, register file, operand collector, and LSU-Extension are derived from cacti [15]. For all the above-evaluated components on the DRAM die, we conservatively assume ×2 area overhead considering the reduced number of metal layers in the DRAM process [76]. For multi-row-buffer support, we include the overhead of 128 extra row address latches [53] per memory controller to enable simultaneous activations of 4 subarray row buffers. For the GPU evaluation, the GPU performance and power results are collected with the help of nvpof and nvidia-smi, respectively.

**B. Performance, Area, Energy, and Thermal Analysis**

**Performance.** MPU achieves 3.45× speedup on average over the GPU as shown in Fig.8 (1). This speedup is contributed by the improved memory bandwidth from the hybrid-pipeline near-bank architecture, the architecture optimizations for the SIMT programming model (Sec VI-C), and the compiler optimizations for the locations of instructions (Sec VI-D).

To further extend different speedup numbers across workloads, we plot the memory intensity (Byte/Instruction) and the speedup of these workloads in Fig.8 (2). First, we observe that the speedup number has a strong correlation with the memory intensity because the memory intensity represents the demand of workloads for memory bandwidth. As MPU provides more memory bandwidth than the GPU (4.13× in measurement), for benchmarks with simple memory access and compute patterns (e.g., AXPY), the speedup is proportional to the memory
Table III

| Name               | Number | Area Per Die (mm²) | Overhead (%) |
|--------------------|--------|-------------------|--------------|
| Shared Memory      | 4      | 0.84              | 0.88         |
| Register File      | 16     | 9.71              | 10.12        |
| Memory Controller  | 16     | 0.63              | 0.66         |
| Operand Collector  | 64     | 2.43              | 2.53         |
| Vector ALU         | 16     | 3.74              | 3.90         |
| LSU-extension      | 16     | 2.43              | 2.53         |
| Multi-row-buffer Support | 64 | 0.04             | 0.01         |
| Total              |        | 19.80             | 10.62        |

Fig. 9. Energy and energy reduction comparison with the GPU.

Area. MPU's hybrid pipeline architecture is area-efficient because only a small part of the pipeline backend components are added in the DRAM die, saving the area for other pipeline units. In Table II, we evaluate the area of added components and normalize the total overhead to a DRAM die (96mm² [68]). Thanks for our compiler optimizations (Sec. VI-D) which significantly reduce the near-bank register usage, we shrink the near-bank register file to half the size of the far-bank register file. This brings the total area overhead from 30.74% to 20.62%. We argue this overhead is small for a general purpose SIMT processor, comparing to 10.71% area overhead in previous work which only supports domain-acceleration [23].

Energy. MPU achieves 2.57× energy reduction on average over the GPU (Fig. 9). The energy reduction mainly comes from the reduction of expensive data movement compared with the GPU, since MPU has a much shorter and simpler data path to access a core’s local DRAM banks. Compared with the complex data path components in the GPU, where the data needs to travel through the TSVs inside the HBM, off-chip links, L2 cache, crossbar network, and then L1 cache to the local register file, the MPU directly offloads the instruction to the DRAM dies to transfer data between the near-bank register file and the DRAM banks. Also, we observe that for each benchmark the energy reduction in Fig. 9 is approximately proportional to the speedup in Fig. 8 (1). This is because MPU’s increased bank-level bandwidth is a result of near-bank data access, which also contributes to the reduction of data movement energy.

In order to further analyze the energy consumption, we provide a detailed energy breakdown in Fig. 10. We discover that most of the energy in MPU (92.94%) is spent on computation (ALU consumes 39.82%), data access (31.90%), and data movement (21.22%). The data access energy contains local register file access (operand collectors (OPC) and register file (RF) consumes 15.47%) and DRAM accesses (16.42%). For data movement, the energy spent on remote data movement (Network consumes 4.43%) is significantly smaller than the local data movement (TSV consumes 16.79%). This well explains the data movement saving advantages of MPU compared to GPU to achieve great energy reduction.

Shared memory optimization. To understand the benefit of our near-bank shared memory, Fig. 11 shows performance results compared with placing the shared memory on the base logic die, denoted as far-bank shared memory. In the same figure, we also plot TSV traffic improvement of near-bank shared memory design w.r.t. far-bank shared memory design. On average, near-bank shared memory design achieves 1.48× speedup and 1.89× TSV traffic improvement compared with...
far-bank shared memory design. The performance benefits of near-bank shared memory come from the extensive use of shared memory. If the shared memory location is far-bank, the contents of near-bank registers need to be brought down to the base logic die for the inter-thread communication through shared memory. This will create a lot of register movement traffic and congest the TSVs. For the near-bank shared memory design, the default locations of value registers for \texttt{ld/st.global} and \texttt{ld/st.shared} are all near-bank. Thus less register movement will be involved, easing the bandwidth pressure on the TSVs. However, since the number of instructions offloaded to NBUs also rises, this may increase TSV traffic, as we observe that for some workloads with speedup larger than 1, the TSV traffic improvement may be slightly less than 1 (HIST, NW). For workloads that do not use shared memory, both the performance and TSV traffic are identical to the location of shared memory.

**Multiple activated row-buffers analysis.** To understand the benefits of multiple activated row-buffers, we compare the performance of all workloads running on MPU with different numbers of activated row-buffers. Fig.12 shows such performance comparisons where the speedup is normalized to a single row-buffer. As shown in the Fig.12 (1), the speedup numbers are 1.10× and 1.25× when we increase the number of activated row-buffers to 2 and 4, respectively. The row-buffer miss rate in Fig.12 (2) indicates that as we increase activated row-buffers to 2 and 4, the miss rate reduces from 15.60% to 9.20% and 5.45%, respectively. Because more activated row-buffers can effectively reduce the row-buffer ping-pong effect in the dynamic scheduling of warps, increasing the number of activated row-buffers effectively reduces average DRAM access latency to improve end-to-end time. Especially, we observe that KNN, UPSAMP, and AXPY significantly benefit from the increased number of activated row-buffers due to severe ping-pong effects on a single row-buffer.

**Comparison with processing-on-base-logic-die (PonB) solution.** We compare MPU with the state-of-the-art general purpose near-data SIMT processors by placing all compute logic on the base logic die, denoted as PonB. The end-to-end execution time shown in Fig.13 demonstrates that on average MPU achieves 1.46× speedup up compared with the PonB solution. This performance improvement is contributed by a significant amount of instructions offloaded for near-bank computations. This reduces data movements on the TSVs which have a much lower bandwidth than bank-level memory bandwidth.

**D. Effectiveness of Compiler Optimizations**

We first conduct static analysis according to the iterative algorithm introduced in Sec.V-B to infer the locations of registers. The breakdown of registers on different locations for all workloads is shown in Fig.14. On average, 32.5% registers
only appear in near-bank locations, 63.7% registers only appear in far-bank locations, and 3.8% registers could appear in both locations. Because only registers appearing in near-bank locations need to use the near-bank register file, we effectively shrink the size of the near-bank register file to reduce its area overhead. This breakdown also demonstrates a clear separation of near-bank registers from far-bank registers. Only a small portion of registers could appear in both locations. This clear separation comes from a clear separation of two classes of dependency chains. The first class involves computations on the data value loaded from the DRAM, and the second class involves integer calculations for DRAM addresses and control-flow related variables, such as loop variables. Usually registers for these two classes of dependency chains do not interfere with each other, as registers associated with the first class usually exist in near-bank locations, and registers related to the second class reside on the base logic die. Therefore, for most registers (more than 95%) we can assign them to a certain near/far-bank location to reduce the register file usage.

We further evaluate the performance of different instruction location policies with the GPU as shown in Fig 15. On average, using the proposed instruction annotation optimizations, we achieve 3.45× speedup w.r.t. GPU. However, using hardware’s default instruction location policy, offloading all instructions to near-bank compute-logic, or offloading all instructions to far-bank compute-logic, we achieve 1.92×, 1.22×, and 1.78× speedup, respectively. Compared with the default hardware policy and both naive offloading strategies, our instruction location annotation is based on the annotated register location. Because of the clear separation of two classes of aforementioned dependency chains, our instruction location annotation assigns most of the computation on data values to near-bank and computation on addresses or control-flow conditions to far-bank. As a result, the register movement traffic on TSVs is minimized, which eventually boosts the performance of programs running on MPU.

VII. Related Work

General Purpose Near-data-processing Platforms: Pioneering studies [18], [30], [46] attempted to integrate the entire processor on the DRAM die, which incurs considerable area overheads. Compared with them, MPU only places lightweight components in the DRAM dies through a hybrid pipeline, which significantly reduces the overhead. Recently, there are a number of practical general purpose near-data-processing solutions that explore near-cache [41], near-memory-controller [57], near-DIMM [7], [10], and 3D-stacking processing-on-logic-die CPU-style [4], [13], [17] and GPU-style [25], [32], [61], [71], [72] platforms. However, these solutions have several drawbacks. First, they have moderate bandwidth improvement, due to the hierarchically shared bus of the main memory. To overcome this drawback, MPU unleashes bank Internal bandwidth through near-bank computing. Second, the communication between the host and the near-data logic introduces extra data traffic because of shared memory space, which may offset the benefit of near-data-processing, including fine-grained instruction offloading overhead [4], [25], cache coherence traffic [13], concurrent host access stall [17], and inconsistent data layout requirement [72]. Different from these prior studies, MPU has an independent memory space and supports end-to-end kernel execution, the same as discrete GPU cards [43], [58].

Domain-specific Near-data-processing Accelerators: A large number of previous studies have explored domain-specific accelerators using near-data-processing ideas, including approximate computing [76], image and video processing [23], [74], deep learning [5], [67], graph analytics [48], bioinformatics [25], garbage collection [28], address translation [62] and data transformation [6]. These designs usually adopt domain-specific processing logic, customized data paths, and application-tailored software mapping strategies. The lack of programmability for these accelerators confines them to a niche application market, adding non-recurring engineering costs for silicon manufacturing. In contrast, the SIMT programming model supported by MPU can benefit a wide range of data-intensive parallel programs, and our end-to-end compilation flow greatly eases the burden of programmers.

Analog Process-in-memory Architecture: In addition to the digital near-data-processing solutions, recently there is a surge in researches about analog process-in-memory (PIM) architecture [47], [70]. Different from the digital solution where the memory array and the computing logic are separate, analog solutions modify the memory array to integrate computing functionalities within memory arrays, thus achieving extremely high computation throughput and energy efficiency. However, these designs suffer from analog noise [38], limited write-endurance issues of non-volatile devices [27], and high overhead of analog digital converters [73]. Although analog PIM solutions are promising for certain application domains such as neural network [16], [60], [69], they are still challenging for general purpose computing usage. In comparison, MPU adopts commercially available 3D stacking technologies [37], [68] without modifying the DRAM bank’s circuit, and this work has demonstrated promising results of MPU on general purpose data-intensive workloads.

VIII. Conclusion

This work proposes MPU (Memory-centric Processing Unit), the first SIMT processor based on 3D stacking near-bank computing architecture. First, we develop a hybrid pipeline where only a small number of hardware components are added on the DRAM dies and the instructions can be offloaded for near-bank computing. Second, we explore two architectural optimizations for the SIMT programming model, introducing a near-bank shared memory design to reduce data movements, and multiple activated row buffers designs to increase bandwidth utilization. Third, we present an end-to-end compilation flow for MPU based on CUDA with a backend optimization to annotate the location of instructions as either near-bank or base logic die through the static analysis of programs. The end-to-end evaluation results of MPU on a set of representative benchmarks demonstrate 3.46× speedup.
and 2.57× energy reduction compared with an NVIDIA Tesla V100 GPU. We further conduct studies to show the performance improvement of MPU over prior 3D-stacking processors and identify the benefits of MPU’s software and hardware optimizations.

REFERENCES

[1] “NVIDIA Tesla V100 GPU Architecture,” 2018, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf [Online]. Available: http://www.nvidia.com

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., “Tensorflow: A system for large-scale machine learning,” in 12th {USENIX} symposium on operating systems design and implementation (OSDI’16), 2016, pp. 265–283.

[3] S. Aga, N. Jayasena, and M. Ignatowski, “Co-ml: A case for co-licerative ml acceleration using near-data processing,” in Proceedings of the International Symposium on Memory Systems, 2019, pp. 506–517.

[4] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture,” in 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). IEEE, 2015, pp. 336–348.

[5] B. Akin and A. R. Alamdeene, “A case for asymmetric processing in memory,” IEEE Computer Architecture Letters, vol. 18, no. 1, pp. 22–25, 2019.

[6] B. Akin, F. Franchetti, and J. C. Hoe, “Hamlet architecture for parallel data reorganization in memory,” IEEE Micro, vol. 36, no. 1, pp. 14–23, 2015.

[7] M. Alian, S. W. Min, H. Asgharimoghadam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O’Halloran, D. Chen, J. Xiong et al., “Application-transparent near-memory processing architecture with memory channel network,” in 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2018, pp. 802–814.

[8] Y. Arafa, A.-H. A. Badawy, G. Chenupati, N. Santhi, and S. Eidenbenz, “Low overhead instruction latency characterization for nvidia gpgpus,” in 2019 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2019, pp. 1–8.

[9] Y. Arafa, A. ElWazir, A. ELKanishy, Y. Aly, A. Elsayed, A.-H. Badawy, G. Chenupati, S. Eidenbenz, and N. Santhi, “Verified instruction-level energy consumption measurement for nvidia gpus,” in Proceedings of the 17th ACM International Conference on Computing Frontiers, 2020, pp. 60–70.

[10] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, “Chameleone: Versatile and practical near-dram acceleration architecture for large memory systems,” in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2016, pp. 1–13.

[11] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, “Analyzing cuda workloads using a detailed gpu simulator,” in 2009 IEEE International Symposium on Performance Analysis of Systems and Software. IEEE, 2009, pp. 163–174.

[12] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, “Near-data processing: Insights from a micro-energ consumption measurement for nvidia gpus,” in Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium, 2012, pp. 33–38.

[13] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, “Prime: A novel processing-in-memory architecture for neural network computation in ternary-based main memory,” ACM SIGARCH Computer Architecture News, vol. 44, no. 3, pp. 27–39, 2016.

[14] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez, “Chonda: Near data acceleration with concurrent host access,” International Symposium on Computer Architecture, 2020.

[15] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang et al., “The architecture of the diva processing-in-memory chip,” in Proceedings of the 16th international conference on Supercomputing, 2002, pp. 14–25.

[16] G. Dupenloup, “Automatic synthesis script generation for synopsys design compiler,” Dec. 28 2004, uS Patent 6,836,877.

[17] Y. Eckert, N. Jayasena, and G. H. Loh, “Thermal feasibility of die-stacked processing in memory,” 2014.

[18] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, “Dynamic warp formation and scheduling for efficient gpu control flow,” in 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) 2007. IEEE, 2007, pp. 407–420.

[19] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Koryzakis, “Tetris: Scalable and efficient neural network acceleration with 3d memory,” in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, 2017, pp. 751–764.

[20] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie, “Pim: Programmable in-memory image processing accelerator using near-back-end architecture,” in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020.

[21] M. Horowitz, “1.1 computing’s energy problem (and what we can do about it),” in 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). IEEE, 2014, pp. 10–14.

[22] K. Hsieh, E. Elharini, G. Kim, N. Chatterjee, M. O’Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, “Transparent offloading and mapping (tom) enabling programmer-transparent near-data processing in gpu systems,” ACM SIGARCH Computer Architecture News, vol. 44, no. 3, pp. 204–216, 2016.

[23] W. Huangfu, X. Li, S. Li, X. Hu, P. Gu, and Y. Xie, “Medal: Scalable dimm based near data processing accelerator for dna seeding algorithm,” in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 587–599.

[24] M. Imani, S. Gupta, Y. Kim, and T. Rosing, “Floatapi: In-memory acceleration of deep neural network training with high precision,” in 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2019, pp. 802–815.

[25] J. Jang, J. Heo, Y. Lee, J. Won, S. Kim, S. J. Jung, H. Jang, T. J. Ham, and J. W. Lee, “Charon: Specialized near-memory processing architecture for clearing dead objects in memory,” in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Computer Architecture, 2019, pp. 726–739.

[26] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shachter, Y. Kim, and W. Sung, “Lada: Transparent offloading and mapping network-on-chip simulator,” in 2013 IEEE international symposium on performance analysis of systems and software (ISPASS). IEEE, 2013, pp. 86–96.

[27] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattanaik, and J. Torrellas, “Flexram: Toward an advanced intelligent memory system,” in Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040). IEEE, 1999, pp. 192–201.

[28] C. D. Kersey, H. Kim, and S. Yalamanchili, “Lightweight sram core designs for intelligent 3d stacked dram,” in Proceedings of the International Symposium on Memory Systems, 2017, pp. 49–59.

[29] G. Kim, N. Chatterjee, M. O’Connor, and K. Hsieh, “Toward standardizered data processing with unrestricted data placement for gpus,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017, pp. 1–12.

[30] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, “A case for exploiting subarray-level parallelism (salp) in dram,” in 2012 39th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2012, pp. 368–379, 44–54.

[31] Y. Kim, W. Yang, and O. Mutlu, “Ramulator: A fast and extensible dram simulator,” IEEE Computer Architecture letters, vol. 15, no. 1, pp. 44–59, 2015.

[32] D. Kirk et al., “Nvidia cuda software and gpu parallel computing architecture,” in ISMM, vol. 7, 2007, pp. 103–104.

[33] P. M. Kogge, “Execube—a new architecture for scaleable mpps,” in Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium, 2012, pp. 33–38.
