# Computing En-Route for Near-Data Processing

Jiayi Huang, *Member, IEEE,* Pritam Majumder, Sungkeun Kim, Troy Fulton, Ramprakash Reddy Puli, Ki Hwan Yum, *Member, IEEE,* and Eun Jung Kim, *Member, IEEE*

**Abstract**—The data explosion and faster data analysis demand have spawned emerging applications that operate over myriads of data and exhibit large memory footprints with low data reuse rate. Such characteristics lead to enormous data movements across the memory hierarchy and pose significant pressure on modern communication fabrics and memory subsystems. To mitigate the worsening gap between high processor computation density and deficient memory bandwidth, memory networks and near-data processing techniques are proposed to keep improving system performance and energy efficiency. In this work, we propose *Active-Routing*, an in-network near-data processing architecture for data-flow execution, which enables computation en-route by exploiting patterns of aggregation over intermediate results. The proposed architecture leverages the massive memory cube- and vault-level parallelism as well as network concurrency to optimize the aggregation operations along a dynamically built *Active-Routing Tree*. It also introduces page granular computation offloading to amortize the offloading overhead and improve the throughput. Compared to the state-of-the-art processing-in-memory architecture, the evaluations show that the baseline *Active-Routing* can achieve up to 7× speedup with an average of 60% performance improvement, and reduce the energy-delay product by 80% across various benchmarks. Further optimizations with vault-level parallelism and page granular offloading can achieve an extra order of magnitude improvement.

**Index Terms**—memory network; data-flow; in-network computing; near-data processing; processing-in-memory.

## 1 Introduction

The amount of data generated has been exploding due to the improvement of technology and advent of numerous network connected devices. This has led to an increasing demand for faster data analysis to extract values from these humongous amounts of data. Thus, data-intensive workloads show very large memory footprints and low data reuse rate. These applications, ranging from machine learning to graph processing [1], [2], have simple computations that operate over myriads of data. Such simple computations in compute kernels and the large amounts of data to be processed trigger considerable data movements across the memory hierarchy. Consequently, modern communication fabrics and memory subsystems face enormous challenges. Moreover, due to the gap between dense CPU computation capability and deficient data supply from memory, computer systems fail to achieve their peak computational power. Thus, it is imperative to drive architectural breakthroughs for reducing data movement to achieve substantial improvement of system performance and energy efficiency.

Recently, tremendous research efforts have been targeted for designing data-centric computer systems. To keep up with the increasing computation power, new memory technologies such as Hybrid Memory Cube (HMC) [3] and High Bandwidth Memory (HBM) [4] provide higher bandwidth by utilizing 3D stacking [5]. Additionally, the traditional processor-centric design is not cost-effective to scale memory capacity and is suboptimal for system bandwidth provision [6]. Therefore, memory-centric designs have been proposed to connect memory modules to form a memory network as a large memory pool and to fully utilize processor and memory bandwidth [6], [7]. These design adoptions may mitigate the data response bottleneck, but still require a significant amount of data movement due to the heavy pressure on the communication fabrics.

Prior research has proposed various optimizations to reduce data movement across the memory hierarchy for better system efficiency. Near-data processing (NDP), as a promising computation paradigm, has propelled architecture advances to move computations near data-resident locations, such as cache [8] and memory [9], [10], [11]. Processing-in-memory (PIM) is an NDP alternative that introduces computation in memory for data processing. Recent studies [12], [13] have proposed to integrate PIM within modern processors in a seamless fashion by extending instruction sets for computation offloading. These mechanisms are most effective in cases of irregular memory accesses and atomic write operations. However, they are suboptimal when performing simple tasks over large amounts of data, such as *dot product*. Extra overhead is incurred by fetching part of the operands across the memory network. Moreover, general purpose PIM typically incurs a large CPU-memory bandwidth overhead for computation offloading through extended instructions.

Previous research has advocated to provide computation power as well as routing functionalities in communication fabrics [14], [15], [16]. The NYU Ultracomputer augmented routers with adders to combine fetch-and-update requests for the same shared variable [17]. To accelerate MPI collec-

- *Jiayi Huang is with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106 USA. E-mail: jyhuang@ucsb.edu.*
- *Pritam Majumder, Sungkeun Kim, Ki Hwan Yum and Eun Jung Kim are with the Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77845 USA. E-mail: {pritam2309,ksungkeun84,yum,ejkim}@cse.tamu.edu.*
- *Troy Fulton is with Aspen Insights, Fort Worth, TX 76107 USA. E-mail: troy.fulton@aspeninsights.*
- *Ramprakash Reddy Puli is with the NVIDIA, Santa Clara, CA 95051 USA. E-mail: rpluli@nvidia.com.*

*Manuscript received 18 Aug. 2020; revised 7 Jan. 2021; accepted 7 Feb. 2021. Date of publication 0 . 0000; date of current version 0 . 0000. (Corresponding author: Jiayi Huang.) Recommended for acceptance by L. Chen and Z. Lu. Digital Object Identifier no. 10.1109/TC.2021.3063378*

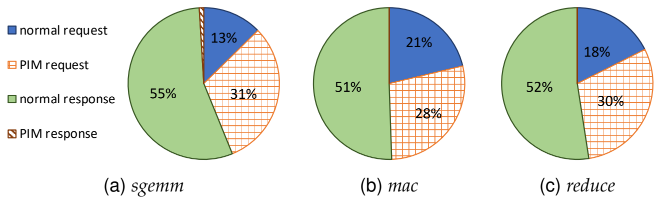


Fig. 1: CPU-mem bandwidth usage breakdown in *Active-Routing* [21].

tives, Panda [18] and Chen et al. [19] proposed reduction units in the network interface to optimize aggregation. These techniques only accelerate pure reduction computations and fail to optimize operations like *dot product*, thereby requiring enormous data movements across the memory hierarchy to calculate the intermediate results before reduction. Recently, MAERI [20] was designed for data-flow computations in deep neural network accelerators to improve efficiency, which does not fit for general workloads. The multiply operations require data to be brought to local SRAM and are computed only at leaf nodes in the tree-based network topology. These in-network computing solutions are limited in flexibility because of the static reduction tree/ring that is tied to the network topology. In this work, we reattempt the communication fabric to facilitate more diverse operations and to build *topology-oblivious* dynamic routing tree for reduction acceleration.

Building upon previous study to further reduce data movement, we have proposed an in-network NDP architecture, *Active-Routing* to enable en-route computing [21]. Compute kernels are mapped to the memory network for data-flow execution by leveraging the aggregation pattern over intermediate results of arithmetic operators. It schedules computations at routers attached to memory in order to exploit the massive memory parallelism and bandwidth. It also builds topology-oblivious *Active-Routing trees* dynamically and takes good use of the network concurrency to optimize reduction operations during the route. We also propose optimizations to exploit both regular and irregular memory access patterns for locality. Despite its effectiveness, there are opportunities for more parallelism and offloading optimizations that we explore in this paper.

Our prior work [21] only uses a single ALU in each cube to process data from many vaults in a cube-level parallel manner, which leaves abundant logic die area for deploying more ALUs at the vault controllers to enable vault-level parallelism for computation throughput improvement. Computing at vault-level not only improves computation throughput, but also reduces the internal bandwidth usage for data communications between vaults and the cube-level *Active-Routing* engine, especially beneficial for pure reduction operations. However, the high overhead of computation offloading hinders the benefit of vault-level parallelism, which is typical in general purpose PIM architectures. As shown in Fig. 1, the offloading traffic in the original *Active-Routing* subscribes up to 30% CPU-memory bandwidth. The offloading traffic behaves as a *many-to-few-to-many* pattern while computations are being offloaded from multiple cores to few memory ports then to many cubes in the memory network. To tackle this problem and unlock the full potential of vault-level parallelism, we propose a page granular offloading optimization to enable offloading in bulk by packing more computation tasks in a single offloading

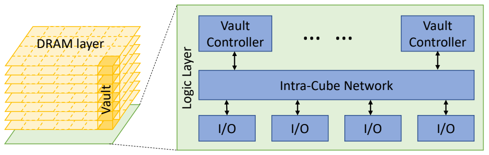


Fig. 2: Hybrid memory cube.

command for regular data that fall within the same page, effectively amortizing the offloading overhead for better efficiency. Building upon our prior work, this paper makes the following contributions:

- We extend the original *Active-Routing* from cube-level parallelism to vault-level parallelism to improve the computation throughput substantially.
- We propose a page granular offloading optimization that offloads computations for regular data that fall within the same page, significantly reducing offloading overhead and improving performance. Note that this technique is not limited to *Active-Routing* and can be applied to other general purpose PIM solutions.
- Our evaluation shows that the baseline *Active-Routing* can achieve up to 7× speedup compared to the state-of-the-art PIM solution; additionally, the proposed new techniques push the performance for an extra order of magnitude improvement.

## 2 Background

### 2.1 Die-Stacked Memory and Processing-in-Memory

Memory technology advances have enabled the fruition of memory and logic integration using 3D stacking [5], making a logic base underneath several stacked DRAM layers. Communications between the logic base and DRAM layers are facilitated by Through-Silicon Vias (TSV), the low-latency and high-bandwidth in-silicon channels. High Bandwidth Memory [4] and Hybrid Memory Cube (HMC) [3] are two popular products of 3D stacked memory. Without loss of generality, we use HMC to demonstrate *Active-Routing* in this work. It is worthwhile noting that it can also be adapted to other interconnects and memory technology such as interposers and HBM. Fig. 2 depicts the HMC structure, which has vertical vault memory partitions that are connected to the logic layer through TSVs. On the logic layer, vault controllers are deployed to manage the vault memory. The controllers are placed sparsely, leaving ample unused silicon budget to implement more complex functionalities. It has been used for implementing computation capability ranging from limited operations [10], [12] to full-fledged processors [9], [11]. HMCs communicate with processor or other memory cubes through four ports. Communications among the vaults and I/O are facilitated by an intra-cube network on the logic die. HMC also enables larger memory size per package and provides abundant internal and external bandwidth with TSVs and high-speed link protocol, which has been leveraged in many existing PIM studies [9], [11].

### 2.2 Memory Network for Scalable Capacity

The limited number of pins per processor chip leads to capacity limits and bandwidth bottleneck in conventional

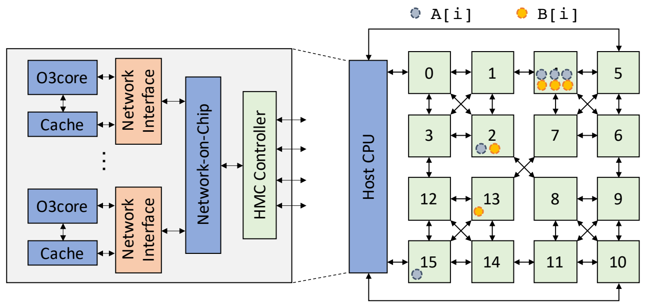


Fig. 3: System configuration with a Host CPU (left) connected to a Memory network (right) that stores vectors A and B.

```
for (i = 0; i < n; i++) {
  sum += A[i] * B[i]
}
```

Listing 1: A *mac* compute kernel for *Active-Routing* illustration.

systems with DDR memory. Hence, more processor sockets need to be installed to scale system memory capacity. However, the bulky data movement with respect to light computation in data-intensive applications can lead to CPU under-utilization due to the deficient data supply. In contrast, HMCs can be interconnected to construct a cost-effective memory network for scalable memory capacity. Also, widely used processor-centric systems improve processor-to-processor communication while overlooking the utilization of system bandwidth. In comparison, memory-centric designs have been demonstrated to achieve better bandwidth utilization in a recent study [6].

### 2.3 Potential of In-Network Computing

Advanced die-stacked memory technology has driven PIM architecture research to realize near-data processing. By providing large memory capacity and high bandwidth, a memory network is adopted to scale PIM architectures to accelerate data-intensive workloads [9]. Computations can be offloaded to the memory network as data-flow graphs, which enable computation en-route along with communication. Hence, further data movement reduction and system efficiency improvement can be achieved through in-network computing by leveraging massive network concurrency and memory-level parallelism.

## 3 Active-Routing Architecture

### 3.1 Architectural Overview

Fig. 3 shows the system configuration where the host CPU is connected to a memory network, which interconnects memory cubes to form a dragonfly topology. In additional, large data vectors `A[]` and `B[]` are stored in several memory cubes for computing a *multiply-and-accumulate (mac)* compute kernel shown in Listing 1. This kernel computes `sum += A[i]`×`B[i]` over a large loop with loop-index `i`. An *Active-Routing* example for this kernel is shown in Fig. 4. In a nutshell, each `A[i]`×`B[i]` computation is embedded in an `Update` packet that is offloaded from the CPU to memory network; then the offloaded `Update` packets compute the local partial sums in the memory cubes through NDP; after offloading `Updates`, a `Gather` packet is issued for the partial results collection from each cube, meanwhile reducing them in the routers on the way back to the CPU. Specifically, *Active-Routing* consists of three processing phases, namely *ARTree Construction*, *Update* and *Gather Phases*.

- While offloading `Update` packets (❶), an *Active-Routing Tree (ARTree)* is being built along their paths toward the scheduled compute memory cubes. For instance, an `Update` packet is sent from CPU via memory cube 0 to cube 4 as shown in Fig. 4a. It records the tree nodes and constructs a tree branch along its path to cube 4. `Update` packets scheduled at different cubes build different branches. Additionally, if vault-level parallelism is applied, a local vault-level branch (with the local vaults as the leaf nodes) is built in each cube. Then all the branches form an *ARTree*, as shown in Fig. 4a.
- Fig. 4b shows the *Update Phase*, during which near-data processing is initiated to drive the offloaded computations. Each `A[i]`×`B[i]` operation needs to fetch its source operands `A[i]` and `B[i]` to complete the computation and update the partial sum in the scheduled cube. Fig. 4b also shows a scenario where two operands are stored in different cubes. In such cases, the `Update` packet is sent to the scheduled compute point that is the last common cube (cube 12) on the minimum routes for both operands. Then, it is replicated to send two requests for `A[i]` and `B[i]` to their resident memory cubes 13 and 15, respectively (❷). Subsequently, two operands are responded to cube 12 to finish the computation (❸). In vault-level parallelism, computation can be dispatched to one of the vaults inside the scheduled cube, and operand responses are routed to the corresponding vault for computation to produce the intermediate result that is aggregated to a partial sum in the local vault.
- The *Gather Phase* sends a `Gather` request after issuing all the `Update` packets as shown in Fig. 4c (❹). It is multicasted from the root to each node of the *ARTree*. Then `Gather` responses at leaf nodes initiate network reduction to aggregate the partial sums computed in the *Update Phase*, which is executed in a data-flow manner from leaf nodes to the root along the *ARTree* (❺). In vault-level parallelism, the partial results in vaults are first reduced in the local cube, then the aggregated partial result is reduced back to the root.

Fig. 5 shows the progress timeline of the three phases in *Active-Routing*. Note that *ARTree Construction* and *Update Phase* can have a large overlap in timeline since the *ARTree* can be built progressively in concurrent with computation.

### 3.2 Three-Phase Packet Processing

*Active-Routing* aims at optimizing reduction by mapping computation kernels to the memory network for data-flow processing. Such a mapping is referred as an *Active-Routing flow*, which is assigned a unique identifier (*flow ID*) for its corresponding *ARTree*. Each *flow* involves a three-phase packet processing procedure as depicted in Fig. 6.

**ARTree Construction.** While processing the `Update` packets, each *flow* builds an *ARTree* dynamically as shown in Fig. 6a. The *flow ID* is registered in the cube when it receives an `Update` packet. If the `Update` packet is not scheduled to

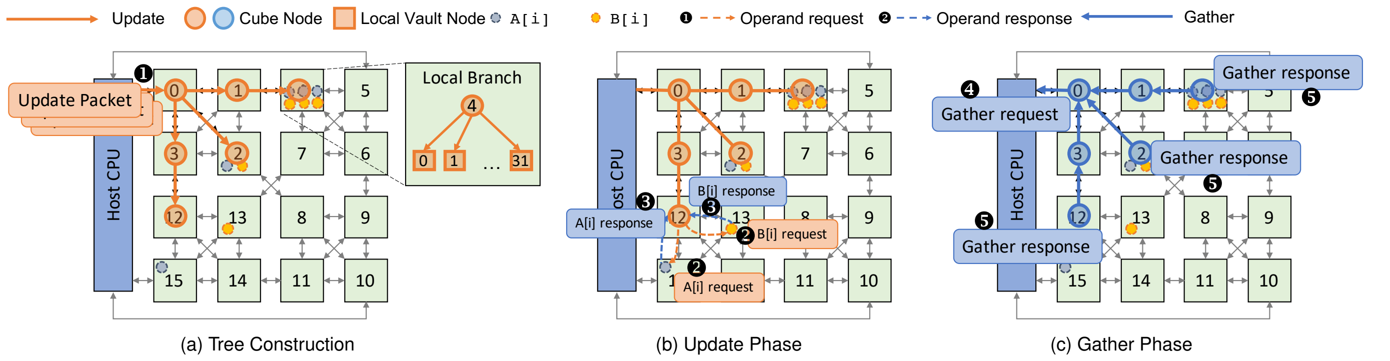


Fig. 4: Active-Routing consists of three phases, including Active-Routing Tree Construction on-the-fly (a), near-data processing in Update Phase (b), and network aggregation along the Active-Routing Tree in Gather Phase (c).

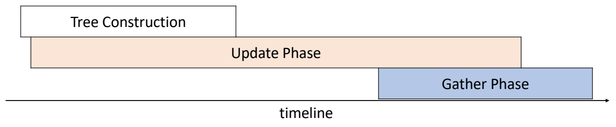


Fig. 5: Active-Routing progress timeline of the three-phase processing procedure.

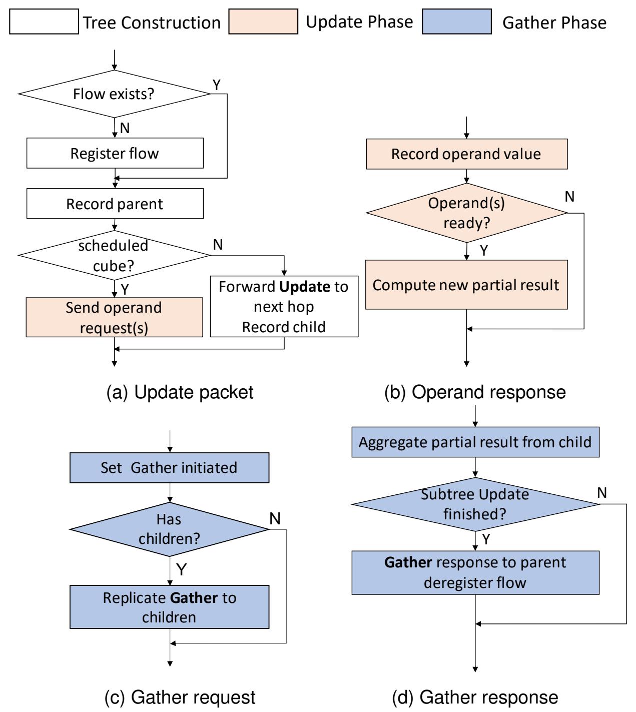


Fig. 6: Packet processing flow chart for update packet (a), operand response (b), gather request and (d) gather response packet (c).

compute at the current cube, it is forwarded to its child based on the routing to its scheduled compute cube. Therefore, by keeping track of parent and children information at each cube, we construct an *ARTree*.

**Update Phase.** NDP is initiated to process `Update` packets and operand request/response packets in parallel with the *ARTree* construction phase, as shown in Fig. 6a and 6b. While processing `Update` packets, operand requests are issued to the memory from the scheduled compute cube. When the operands are responded, the computation is scheduled to calculate the partial result.

**Gather Phase.** Fig. 6c and 6d show the packet processing in *Gather Phase* to commit an *Active-Routing flow*. This phase multicasts the `Gather` requests from the root to leaf nodes in a forward pass, and reduces the partial results from leaf nodes to the root node in the backward pass. Once the subtree of a node completes *Update Phase*, it responds to the parent and releases the *flow* record. After receiving `Gather` responses from all the children, the parent finishes the *Update Phase* of its branch. When the root node completes its own *Update Phase* and receives the `Gather` responses from its children, it commits the *flow* back to the CPU.

## 3.3 Extending to Vault-Level Parallelism

The original *Active-Routing* constructs cube-level trees where each cube maintains the parent-children relationship between the adjacent cubes. In this work, we further extend it to build vault-level trees, allowing vaults to serve as the leaf compute nodes of the *ARTree* as shown in Fig. 4a. In vault-level *Active-Routing*, the local vaults form a local branch while the branch between cubes is a remote branch. Thus, each cube not only maintains the remote branch between cubes, but also keeps track of the local branch for the relationship of the cube (parent) and local vaults (children). With the vault-level parallelism extension, *Active-Routing* can exploit the high TSV bandwidth between DRAM layers and vault controllers as well as reduce the bandwidth usage of intra-cube networks, leading to higher parallelism and better computational throughput. Note that the packet processing procedure presented in the previous section also applies to vault-level *Active-Routing*.

## 3.4 Offloading Optimizations

Due to metadata in the packet header and packet's internal fragmentation in packet-switching techniques, operand fetching and instruction offloading can incur extra overhead. To amortize this overhead, we can take advantage of the memory access patterns of operand fetching by offloading multiple operations at a time. *Active-Routing* targets for optimizing the reduction operation over myriads of intermediate results of arithmetic operators, such as `sum` $= \sum_{i=1}^{n} *A_i \times *B_i$, where $A_i$ and $B_i$ store the operand addresses. When vector `A` stores data addresses of graphs or sparse matrices, the access pattern tends to be irregular. When it stores the array element addresses, the memory access pattern is regular. Hence, the combinations of memory access patterns of the two operands can be grouped into three categories — *irregular-irregular*, *regular-irregular*, and *regular-regular*. Similarly, for pure reduction, the memory access patterns can be simply categorized as *regular* and *irregular*. Based on these categories, we propose several ways to leverage data locality.

### 3.4.1 Cache Block Granular Offloading

We exploit data locality by offloading computations in cache block granularity for *regular* and *regular-regular* access patterns as parallel vector processing. To handle *regular-irregular* access pattern, the irregular data is first fetched and offloaded with computation to the regular data resident location for execution. These two offloading techniques benefit from data locality and effectively reduce memory accesses. Since *irregular* (*irregular-irregular*) memory access pattern breaks data locality, each offloaded packet only contains computation for a single element (pair) as scalar operations. *Active-Routing* can cooperate with previous study [11] to further optimize *irregular-irregular* access pattern, which we leave for future work.

### 3.4.2 Page Granular Offloading

While offloading from multicore CPU through the memory controllers to many memory cubes, the offloading traffic behaves as a *many-to-few-to-many* pattern. Such a pattern can make the memory port a potential bottleneck for offloading, and lead to computation under utilization in the memory cubes. Although cache block granular offloading can process the data faster with vector processing, the computations embedded in an `Update` packet is limited. Therefore, packing more computations in each offloaded packet can increase the computation request intensity to improve utilization and throughput. In applications with *regular* memory access patterns, the data to be processed are usually in kilo-bytes or even mega-bytes with sequential addresses, which can be many pages of memory. Fortunately, each physical page frame mapped by a virtual page resides in a single cube, so the `Update` requests for data in the same page are issued to the same cube. To exploit this, we propose *page granular offloading* to pack more computations in an `Update` packet to process consecutive addresses within the same page, aiming at mitigating the offloading bottleneck to work with vault-level parallelism in synergy.

## 4 IMPLEMENTATION

### 4.1 Programming Interface and ISA Extension

We design simple programming interfaces, `Update` and `Gather`, to translate the program semantics into extended instructions. The ISA extensions are used to communicate with network interfaces to offload computations to the memory network for *Active-Routing* execution.

```
UpdateRR(void *src1, void *src2, void *target, int op);
UpdateRI(void *src1, void *src2[], void *target, int op);
UpdateII(void *src1, void *src2, void *target, int op);
UpdatePage(void *src, int num_lines, void *target, int op);
UpdateRRPage(void *src1, void *src2, void *target, int
    num_lines, int op);
Gather(void *target, int num_threads);
```

Listing 2: Active-Routing programming interfaces.

Listing 2 shows the definitions of `Update` and `Gather` APIs that facilitate offloading of *Active-Routing flows*. The first three `Update` APIs carry two source memory addresses of an arithmetic operation. The postfix `RR`, `RI` and `II` of `Update` APIs are used for *regular-regular/regular*, *regular-irregular* and *irregular-irregular/irregular* memory access patterns, respectively. The `UpdatePage` API is designed to support page granular offloading for *regular* pure reduction, where the `src` argument is the base address for the consecutive cache lines within the same page for the operands, and the `num_lines` indicates the number of consecutive cache lines to be processed. Similarly, the `UpdateRRPage` API is introduced to support page granular offloading for *regular-regular* two-operand computations. Note that the allocated memory for both the operands should be aligned to page size. In the best case, the `src/src1/src2` is the base address of a page and `num_lines` is the total number of cache lines in a page, which offloads computation for a full page. The opcode is passed through the `op` argument to indicate the arithmetic and reduction operation, such as float multiply-and-accumulate. The `target` field in the APIs is the address of the reduction variable that is hashed to a unique identifier for each *flow*. In `Gather` API, the `num_threads` argument is for the number of threads working on the *flow*, which is used for an implicit barrier at the root of *ARTree* to guarantee all the `Updates` have been initiated. Compilers translate these APIs to extended intrinsic instructions. During execution, instructions are decoded to write the offloading information to a set of dedicated registers in the network interface (NI). Then NI assembles this information into an `Update` or a `Gather` packet and sends it to the memory network. Listing 3 shows the page granular offloading *Active-Routing* code example for the *mac* compute kernel, assuming both `A[]` and `B[]` are page aligned and covers multiple pages of memory. As shown the number of `Update` packets is the same as the number of pages vector `A[]` or `B[]` covers.

```
// A[] and B[] are page aligned and cover multi-pages
int stride = PAGE_SIZE / sizeof(float);
int num_lines = PAGE_SIZE / CACHELINE_SIZE;
float sum = 0;
for (i = 0; i < n; i += stride) {
  UpdateRRPage(&A[i], &B[i], &sum, num_lines, FMAC);
}
Gather(&sum, 1);
```

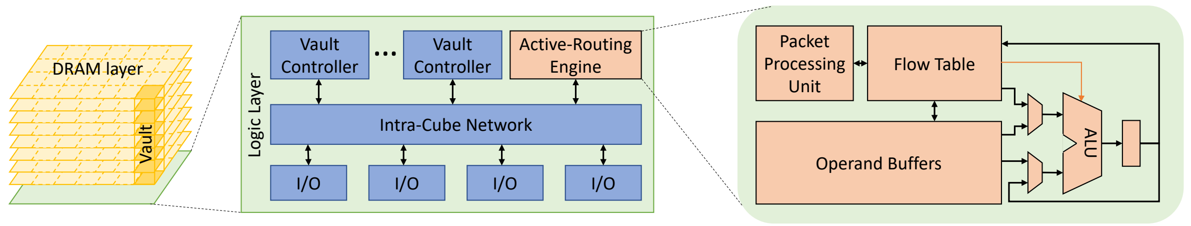
Listing 3: The page granular offloading *Active-Routing* code of *mac*.

### 4.2 Network Interface

*Active-Routing* uses programming interfaces in applications to offload computation. The APIs are translated by compilers into extended intrinsic instructions. These instructions are decoded to assemble packets that are offloaded to the memory network. This logic can be added with marginal changes by augmenting the network interface (NI), which connects the processor core to the network-on-chip. In NI, a set of dedicated registers are added to be used by the extended instructions that write opcode and operand information to these registers during execution. Then, these registers are read by NI to compose an `Update` or a `Gather` packet to offload to the memory network.

### 4.3 Active-Routing Engine

On the logic layer of HMC, an *Active-Routing Engine (ARE)* is deployed to facilitate *Active-Routing* functionalities as shown in Fig. 7a. It is integrated as an attached module



(a) HMC logic layer for Active-Routing engine

| 64-bit | 6-bit | 64-bit | 64-bit | 64-bit | 2-bit | 4-bit | 1-bit |
|---|---|---|---|---|---|---|---|
| flowID | opcode | result | req_counter | resp_counter | parent | children flags | Gflag |

(b) Flow table entry

| 64-bit | 6-bit | 64-bit | 1-bit | 64-bit | 1-bit |
|---|---|---|---|---|---|
| flowID | opcode | operand1 | op1_ready | operand2 | op2_ready |

(c) Operand buffer entry

Fig. 7: Active-Routing microarchitecture has the engine implementation in logic layer (a) with flow table entry (b) and operand buffer entry (c).

to the router switch. ARE consists of 1) a packet processing unit to process and generate packets, 2) a *flow table* to keep track of *Active-Routing flows*, 3) a pool of *operand buffers*, 4) an ALU for computation.

### 4.3.1 Packet Processing Unit

`Updates` and `Gathers` are decoded by the packet processing unit, which also schedules corresponding actions as shown in Fig. 6. It generates operand requests to fetch the data and `Gather` response to commit the partial result to its parent.

### 4.3.2 Flow Table

Flow table keeps track of both the states and structure information of each *flow*, whose entry is shown in Fig. 7b. Each table entry is record for a tree node for maintaining the tree structure by holding a unique `flow ID`, an `opcode` for computation, and its `parent` and `children` relationship. It also keeps the state of the *flow*, including the partial `result`, the `req_count`, the `rep_count`, and the `Gflag`. The `req_count` and `rep_count` counter values are used to keep track of the number of issued requests and committed operations. When these two counters are the same, an *Update Phase* is considered finished. The `Gflag` is set by a `Gather` request to indicate that the flow can begin reduction once *Update Phase* is done.

### 4.3.3 Operand Buffers

`Update` packets are processed to generate request(s) to fetch operands and perform the computation after receiving the response. Operand buffer is provided to temporarily hold the operands that are waiting for computation, therefore keeping the pending `Update` operations. We make the operand buffers a shared resource by different flows in order to improve the utilization. An operand buffer entry is reserved before issuing operand request(s), because co-existing *flows* can easily lead to deadlock due to wait-and-hold condition, especially for two-operand operations. An operand buffer entry is shown in Fig. 7c, which holds the `flowID`, the `opcode`, two `ready` flags for the two `operand` fields to indicate their readiness. We use a free and a ready queue to keep IDs of free and ready operand entries, respectively, for ease of direct lookup, thereby reducing the operand buffer access time.

### 4.3.4 ALU

A light-weight ALU is implemented in ARE for arithmetic computations. Different operations over various data types, such as reduction operations sum, division and multiply-accumulate over single- and double-precision floating-point data, can be supported. Table 1 lists the supported operations with their delay and power, which are synthesized using TSMC 28nm library targeting at 1250 MHz clock rate.

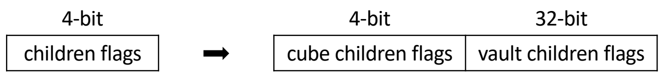
TABLE 1: Delay and Power of Supported Operations.

| Mnemonic | Operation Description | Delay (ns) | Power (mW) |
|---|---|---|---|
| FADD | *float*† add | 0.78 | 1.0449 |
| FMULT | *float* multiply | 0.78 | 1.1316 |
| FMAC | *float mac* | 1.56 | 1.1316 |
| FMAC16 | 16-element vector *float mac* | 3.12 | 18.1056 |
| DADD | *double*† add | 0.77 | 1.9549 |
| DMULT | *double* multiply | 0.78 | 2.7459 |
| DDIV | *double* division | 0.78 | 2.6349 |
| DDAC | *double divide-and-accumulate* | 1.55 | 2.6349 |
| DMAC | *double mac* | 1.55 | 2.7459 |
| DMAC8 | 8-element vector *double mac* | 3.09 | 21.9672 |

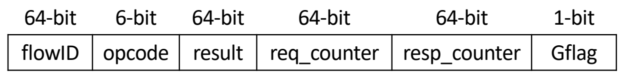
† *float/double* stands for single/double-precision floating-point.

### 4.3.5 Putting It All Together

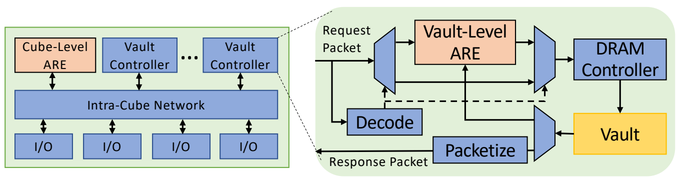
When an `Update` request packet is received, the Packet Processing Unit decodes and processes it. If the corresponding *flow* is not found in the *flow table*, an entry is allocated for the new *flow*. Then the flow is registered and entry fields are initialized by keeping the *flow ID* and the packet's previous hop as parent in the entry. If the packet is not scheduled for the current cube, it is forwarded based on the routing to the next node, which is recorded in the `children flags`. Otherwise, the `req_count` is incremented and an operand buffer entry is allocated from the free queue. Meanwhile, operand request packets are generated with buffer entry ID and the operand address embedded. If all the operand buffers are occupied, the packet processing unit is stalled until an entry becomes free. When the operand response arrives, its operand buffer entry is updated. If operands are ready, the operand entry ID is pushed to the ready queue for processing. ALU inspects the ready queue to schedule computation. After finishing execution, `result` is updated and the `resp_count` is incremented in the corresponding *flow* entry. The operand buffer is then released and its ID is pushed back to the free queue for reuse. While processing `Gather` request packets, the `Gflag` of the corresponding flow table entry is set to initiate *Gather Phase* after the completion of the *Update Phase* of the subtree. If the cube has children cubes, the packet is replicated and sent to its children. Upon receiving a `Gather` response from a child for partial result update, its corresponding child field is cleared. Note that



(a) Cube-level flow table entry



(b) vault-level flow table entry



(c) Augmented vault controller for Active-Routing support

Fig. 8: Vault-level parallelism support with cube-level flow table entry (a), vault-level flow table entry (b), and augmented vault controller (c).

every time the `result` is updated by either computation in the current cube or `Gather` packet from a child. If the `children flags` are cleared and the `Gflag` is set, a `Gather` packet is generated to reply to the partial result to the parent and release the flow table entry.

## 4.4 Vault-Level Parallelism Support

With vault-level parallelism extension, the *ARE* in the original *Active-Routing* now is termed cube-level *ARE* for differentiation. To support vault-level *Active-Routing*, the cube-level *ARE* acts as the parent for all the local vault controllers and issues `Update` packets to the vault children. The cube-level flow table entry is extended to track the local vault children as depicted in Fig. 8a. For simplicity, the cube-level *ARE* dispatches `Update` packets to the vaults in a round-robin fashion for *regular-regular* two-operand computations. For pure reduction, it issues `Updates` to the local vaults based on the operand address to reduce inter-vault communication. Note that unlike the original *Active-Routing*, vault-level parallelism designates the cube-level *ARE* only as a reduction point for the `Gather` responses from the cube and vault children, rather than as the compute point for all data in the cube. This also simplifies the design of ALU and operand buffer. The vault controller is augmented with a compute engine as shown in Fig. 8c, where the vault-level flow table entry is simplified because the vault controller is the leaf node and always has the cube-level *ARE* as the parent. Once a packet arrives at the vault controller, the decoder inspects the header of the packet and directs it to the vault-level *ARE* if the packet is either an `Update` request or an operand response from a remote vault. The local data response from the vault is also directed to the local vault-level *ARE* if it is a local data response. The workflow of *ARE* remains the same as in cube-level parallelism. A credit-based flow control of operand buffer availability in the vault-level *ARE* to the cube-level *ARE* is maintained for computation scheduling to avoid protocol deadlock and guarantee correctness.

## 4.5 Page Granular Offloading Support

As data in a page is distributed in different vaults in a cube, it is expensive to process the page granular `Update` packet as page-wide vector processing. Therefore, we generate cache block granular `Update` packets from a page granular `Update` packet to leverage the cache block vector processing. The number of generated cache block granular packets depends on the `num_lines` (the number of consecutive cache blocks) embedded in the page granular `Update` packet. When the original *Active-Routing* is enhanced with page granular offloading optimization, the *ARE* processes the page granular `Update` and generates one cache block granular `Update` packet at a time. Meanwhile, it decrements the `num_lines` and updates the addresses information in the packet header. When `num_lines` is zero, cache block granular `Updates` for all the offloaded cache blocks have been issued and the page granular `Update` packet is consumed. Similarly, in VLP, the cube-level *ARE* processes the page granular `Update` and dispatches the computations to the vaults. Then each vault controller fetches the operand cache block(s) from either local or remote vault to finish the dispatched computation.

## 4.6 Integrity Considerations

Virtual memory support and cache coherence are two important considerations for seamlessly integrating *Active-Routing* into modern computer systems. In this section, we describe how they are supported to enable *Active-Routing*.

### *4.6.1 Virtual Memory*

As *Active-Routing* is implemented by extending the ISA, the offload instructions are treated as extended *active* loads/stores. Therefore, they can perform the same virtual to physical address translation as normal load/store instructions. For the page granular offloading, since all the data in a page resides in the same physical frame, it is also supported by modern virtual memory systems. With this design principle, we can avoid overhead for address translation units in the directories, or memory.

### *4.6.2 Cache Coherence*

To offload *Active-Routing* instructions, it is important that the offloaded *flow* is using the up-to-date data in memory. A naïve way is allocating an uncacheable memory region for the data that is used by *Active-Routing*. However, it can hurt the performance in other program execution phases which can use the deep cache hierarchy to exploit locality. To ensure coherent *Active-Routing*, offloaded packets are first sent to the directory to query for back-invalidation if data is cached on-chip similar to [12]. Then it is issued to the memory. Since `Update` packets are sent in parallel, the back-invalidation overhead is amortized across massive concurrent packets. We observe that back-invalidation happens rarely in our experiments. Further optimizations can be applied by integrating a recent coherence mechanism dedicated for near-data architectures [22].

## 4.7 Enhancements in Active-Routing

We have two observations that may significantly impact the performance of *Active-Routing*: (1) the decision for choosing the root of a tree can affect the network congestion, and (2) the offloading overhead widely varies with the change in its granularity. To further improve *Active-Routing* performance, we address these two points as follows.

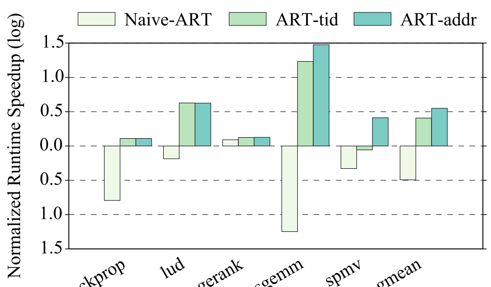


Fig. 9: Runtime speedup of ART variants over HMC Baseline.

Since the computations are offloaded from the host CPU through the memory ports, we naturally consider the cubes that are attached to the four channel ports as root node candidates. Starting from a static approach, cube 0 is always assigned as the root node. In order to balance the load better in the network, we propose two enhancements that consider all the four corner cubes as root candidates and are able to create multiple trees for one *flow*. The first one uses thread ID to interleave the candidate cubes in order to balance the trees rooted from four corners among multi-thread applications, named as ART-tid. Since the scheduling is oblivious to the data location, it can create deep trees and lead to more hop traversals for `Update` request packets. Another enhancement technique takes the operand addresses into account and sends the `Update` packet through the port nearest to its destination. This creates shallow trees with respect to ART-tid, we name it as ART-addr. When massive packets are created, it may lead to congestion on certain ports depending on memory addresses. Since these two schemes can create multiple *ARTrees* for one *flow*, the extended HMC memory controllers that manage the trees are coordinated to merge the subflows at the end of *Gather Phase*. On the contrary, Naïve-ART constructs only one *ARTree* for each flow.

To reduce offloading overhead and the number of memory accesses, we adapt the offloading granularity, to exploit the data locality of different memory access patterns discussed in §3.4. This optimization is applied to both ART-tid and ART-addr, whereas Naïve-ART neither considers granularity nor data locality and simply offloads every single operand pair. This Naïve-ART may experience contention in operand buffer resources, and network contention in addition to high offloading overhead due to static manner for tree construction and simple offloading.

Fig. 9 shows the improvement impacts of the enhancements over Naïve-ART in log scale speedup that is normalized to HMC conventional system baseline (not shown). It shows that with the naïve way of static tree formation and offloading, Naïve-ART is even worse than HMC baseline, especially when there is some data locality. In contrast, better performance can be achieved by exploiting the memory access patterns and building the trees dynamically. In the following sections, we only present ART-tid and ART-addr for detail analysis.

TABLE 2: System Configurations

| Parameter | | Configuration |
|---|---|---|
| CPU | Core | 16 OoO cores @ 2GHz<br>issue/commit width: 4, ROB: 128 |
| | L1I/D Cache | Private, 32KB, 4 way |
| | L2 Cache | S-NUCA 16MB, 16 way, MESI |
| | NoC | 4x4 mesh, 4 MC at 4 corners |
| Memory | DRAM Timing | $t_{CK} = 0.8$ ns, $t_{RAS} = 21.6$ ns, $t_{RCD} = 10.2$ ns<br>$t_{CAS} = 9.9$ ns, $t_{WR} = 8$ ns, $t_{RP} = 7.7$ ns |
| | HMC | 4GB/cube, 4 layers<br>32 vaults, 8 banks/vault |
| | HMC Network | 16 cube DragonFly, 4 controllers<br>Minimal routing, virtual cut-through<br>16 lanes link, 12.5 Gbps/lane<br>CrossbarSwitch clock @ 1250 MHz |
| Active-Routing Engine | Flow Table | 16 flow entries |
| | Operand Buffer | 128 buffer entries |
| | Processing Element | 1250 MHz clock frequency<br>An arithmetic logic unit |

## 5 Methodology

### 5.1 System Modeling and Configuration

We use an execution-driven simulator McSimA+ [23] with detailed microarchitecture models as the backend for cores and cache hierarchy. We integrated a cycle-accurate simulator CasHMC [24] with McSimA+ to replace its memory system for HMC memory modeling, which is further extended for interconnect functionality. We leveraged McSimA+'s Pin [25] based front end to implement *Active-Routing* instruction extensions. The microarchitectural behaviors of *Active-Routing* were implemented in the crossbar switch and vault controllers on the HMC logic layer. We configured the host CPU as a chip-multiprocessor with network-on-chip and two-level cache hierarchy with MESI coherence protocol. The 16 off-chip HMCs are connected in a Dragonfly topology [6]. The system configuration evaluated in this work is shown in Fig. 3 and described in Table 2.

For power modeling, we use CACTI [26] for CPU on-chip cache and HMC buffer power estimation, assuming 5pJ/bit for each hop in memory network [27] and 12 pJ/bit for HMC memory access [3]. We conservatively model 64-byte size for both a flow table entry and an operand buffer entry due to the parameter constraint of CACTI. The access time, energy and area for the flow table are 0.428 ns, 0.0620 nJ/read (0.0626 nJ/write) and 0.069 mm$^2$, respectively. The access time, energy and area for the operand buffers are 0.352 ns, 0.0342 nJ/read (0.0474 nJ/write) and 0.252 mm$^2$, respectively. The ALU is implemented in Verilog and synthesized using TSMC 28 nm library. The timing of different operations are listed in Table 1. The power and area of the ALU is 43.05 mW and 0.15 mm$^2$, respectively. Note that *Active-Routing* does not incur extra computation energy but just changes the place of computation. For vault-level parallelism support, extra ALUs and flow tables are required for each vault while operand buffers can be distributed, which accounts for 7.01 mm$^2$ of 32 more ALUs for 32 vaults in a memory cube.

### 5.2 Workloads

*Active-Routing* targets applications that have abundant reduction on data processing operations such as multiply-accumulate or pure reduction operations over a large memory footprint. We studied five kernels from several benchmark suites. These kernels are widely used in diverse appli-

TABLE 3: Workloads

| Workloads | | Optimization Region | Input Data Size |
|---|---|---|---|
| Benchmarks | *backprop* [28] | activation calculation in feedforward pass | 2097152 hidden units |
| | *lud* [28] | upper and lower triangular matrix decomposition | 4096 matrix dimension |
| | *pagerank* [2] | ranking score calculation | web-Google graph [29] |
| | *sgemm* [30] | matrix multiplication | 4096x4096 matrix |
| | *spmv* [30] | matrix-vector multiplication loop | 4096 matrix dimension and 0.7 sparsity |
| Microbenchmarks | *reduce* | sum reduction over a sequential vector | 6400K dimension |
| | *rand_reduce* | sum reduction over random elements | 6400K elements |
| | *mac* | multipy-and-accumulate over two sequential vectors | two vectors with 6400K dimension |
| | *rand_mac* | multiply-and-accumulate over two random element lists | two lists with 6400K elements |

cation domains such as scientific computing, graph analytics, language modeling and deep learning. We also develop four data-intensive microbenchmarks for case study. In order to support execution with McSimA+ frontend, all the applications were re-implemented using the Pthread library. We chose sufficient large input data so as to stress the last level cache and memory as well as to account for reasonable simulation time. The working set sizes varied from 80 MB to 0.5 GB. The memory requirements of these kernels used in various applications tend to grow significantly larger as data scales.We summarize the workloads and applied optimization region as well as input data in Table 3.

## 6 Evaluation

In this section, we evaluate ART-tid and ART-addr and compare them to PEI [12], which is implemented by adding a computation unit in each vault controller to support PEIs. It can compute a dot product of two 4-dimension vectors in a cycle, one of the vector operands (either regular or irregular) are brought to cache and sent to the memory location of the other half (should be regular) for processing in memory. We first compare the performance followed by power and energy analysis. Then we show the impact of vault-level parallelism and offloading granularity, as well as the potential of dynamic offloading through a case study.

### 6.1 Performance

#### 6.1.1 Speedup

Fig. 10a and 10b show the runtime speedup of benchmarks and microbenchmarks, respectively. Both ART-tid/addr schemes form multiple trees from all the memory ports for massive *flows* in the benchmarks. The results show more than 6% performance improvement across various applications with respect to PEI except *lud*. Specifically, ART-addr improves *sgemm*, a dense matrix multiplication kernel by 7×. In *sgemm*, almost all the runtime is spent in matrix multiplication. During execution, PEI needs to fetch one of the source matrices and also update the target matrix, which causes read-write contention on the cache, leading to cache thrashing. In contrast, ART has no contention between source matrices and target matrix since both source matrices are processed in memory, thereby outperforming PEI. In geomean, ART-tid and ART-addr improve performance by 15% and 60% over PEI, respectively. For *lud*, PEI performs slightly better than both ART-tid and ART-addr. In case of *spmv*, PEI outperforms ART-tid but performs worse than ART-addr. This is because in these two applications, the computation distribution is not balanced, which causes contentions in compute/buffer resources.

Note that the PEI implementation is optimistic since we have no limit on operand buffers. For *spmv*, ART-addr works better than ART-tid because of more balanced work distribution. In microbenchmarks, the whole execution is the region of interest for optimization. Both ART-tid/addr alternatives work well across all microbenchmarks. Compared with PEI, ART-tid/addr achieves 7×/10× speedup, respectively.

Fig. 13 shows a heatmap of *spmv* for ART-tid and ART-addr. In the heatmap darker colors are used for denoting higher number of event occurrences. Each big square depicts the whole memory network and each small square block represents one cube in the memory network. In ART-addr, the work is evenly scheduled in each cube which can have better resource utilization. While in ART-tid, computations are centered in a few cubes which leads to compute/-operand resources contention and less parallelism[1].

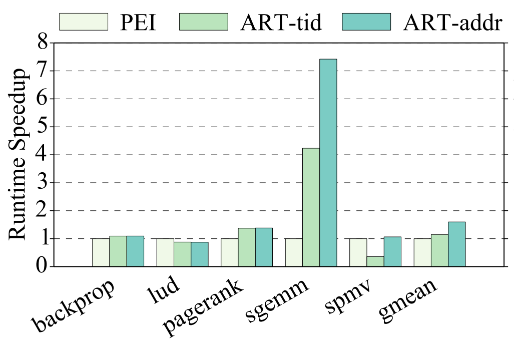
#### 6.1.2 Update Offloading Round-trip Latency

Fig. 11 shows the round-trip latency breakdown into request, stall and response for understanding the contribution of different communication components of `Update` offloading. As expected, the total latency is inversely proportional to the performance shown in Fig. 10. In general, ART-tid and ART-addr dynamically distribute the `Updates` across all available ports for tree construction. The ART-tid/addr schemes can balance the load evenly and utilize the memory network resources more efficiently. Compared to ART-tid, ART-addr has lower round-trip latency across all benchmarks. ART-tid constructs trees by interleaving memory ports using thread IDs. Therefore, the tree root is not necessarily close to the directory where `Update` packets check for coherence. In contrast, ART-addr distributes `Updates` based on addresses, which makes the tree root physically close to the directory, thereby incurring less request latency. The stalls are mostly due to queuing in HMC controllers.

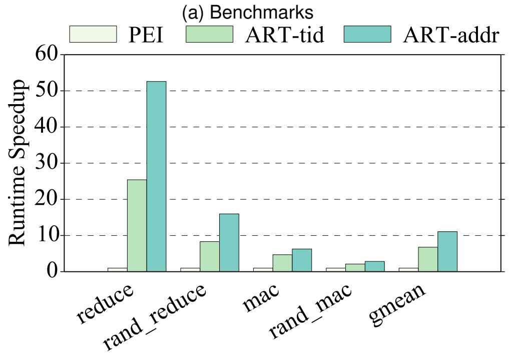
#### 6.1.3 Data Movement

We evaluate data movement as the data size transferred between the host processor and memory network. Fig. 12 shows the data movement breakdowns for normal data and active data transfer. For most applications, ART-tid/addr can reduce the memory requests compared to PEI. In *pagerank*, the region of interest for optimization is the code segment that has reduction on large amounts of data processing tasks. In this benchmark, only parts of the whole parallel phase can be accelerated by *Active-Routing*. The other phases still require data movement. Another overhead comes from

1. The operand distribution are different due to the dynamic memory allocation.

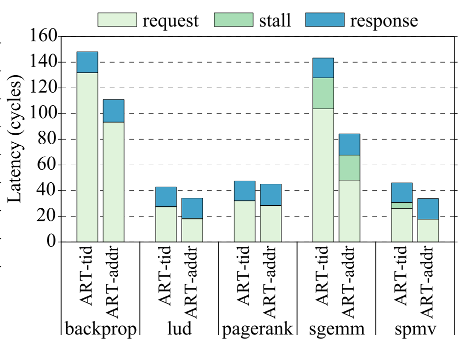


(a) Benchmarks

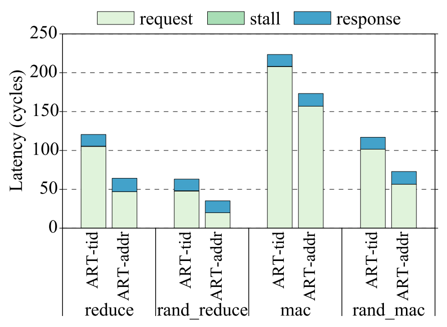


(b) Microbenchmarks

Fig. 10: Runtime speedup over PEI.

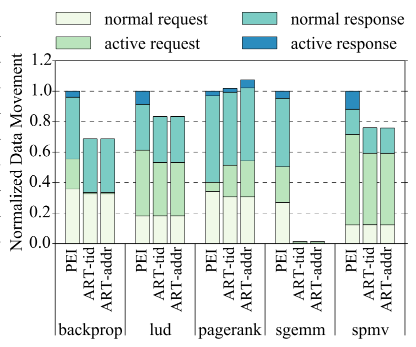


(a) Benchmarks

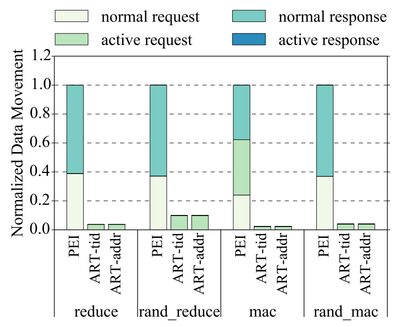


(b) Microbenchmarks

Fig. 11: Update round-trip latency breakdown into request, stall and response latency.



(a) Benchmarks



(b) Microbenchmarks

Fig. 12: On/off-chip data movement normalized to PEI.

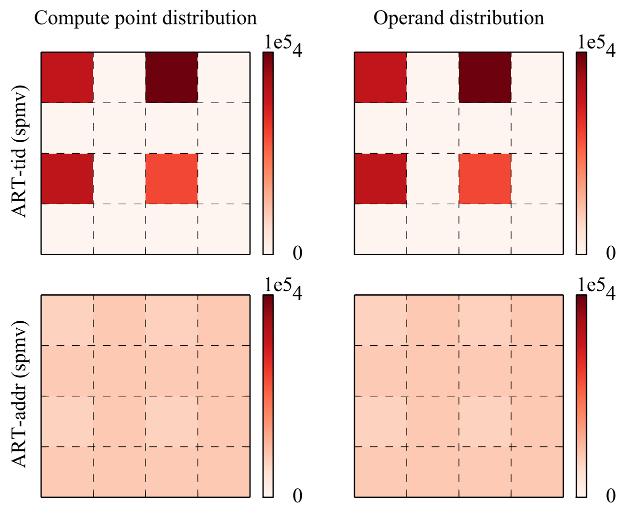


Fig. 13: SPMV compute point and operand distribution.

massive fine-grained offloading due to the irregular memory access pattern.

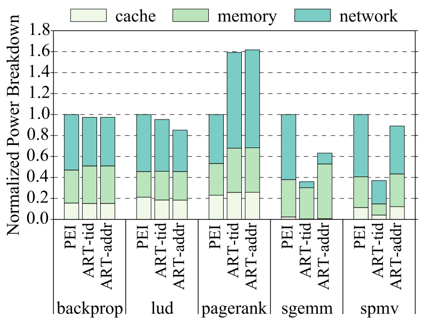
In the microbenchmarks, the whole parallel phase can be optimized, so the data movement decreases significantly. In *reduce*, the majority of the runtime is spent on summing up all the elements of a large array as it accesses the array elements sequentially. Similarly, *mac* operates multiply-and-accumulate over two large vectors. Both of them have friendly spatial locality in their memory accesses, which is exploited in cache-block granular offloading. However, PEI needs to bring part of the data on chip and offload it with the instruction, causing data movements. For *rand_reduce* and *rand_mac*, ART-tid/addr have more data movements compared to the sequential accesses due to offloading overhead. Since PEI still needs to bring the data for random multiplication on chip before atomic write, it incurs more data movement.

## 6.2 Power and Energy

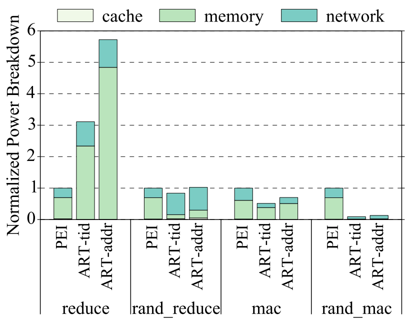
### 6.2.1 Power Consumption

We present the power consumption breakdowns into cache, memory and memory network in Fig. 14, which shows that ART-tid/addr consumes similar memory power and less network power than PEI except for *pagerank*. In ART-tid/addr, data is fetched from memory and communicated in the network. However in PEI, part of the operands need to be brought across the network to on-chip cache and be sent with the offloaded instruction, leading to cache contention even cache thrashing. For example, *sgemm* has cache contention between reading of large source matrix and writing to target matrix. The cache thrashing leads to more memory accesses. Consequently, PEI and ART have similar memory access intensities. For regular memory accesses in terms of network power, ART feeds the data in the network with minimum routing while PEI brings data all the way to the CPU, thus PEI consumes more power. One exception is *pagerank* that has irregular memory access patterns. ART offloads computation *flows* in single operand granularity, causing high overhead in offloaded packets and operand packets, thereby consuming more power.

Microbenchmark *mac* has similar power characteristics as the benchmarks that have regular memory access patterns. For *reduce*, ART-tid/addr can massively process the reduction near-data in memory cubes without moving data around, which leads to more intensive memory accesses and more offloading. For irregular memory access patterns such as *rand_reduce* and *rand_mac*, PEI has no data reuse and can only optimize atomic updates, leading to many memory accesses with more power consumption.

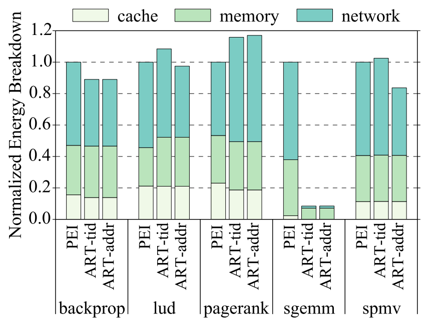


(a) Benchmarks

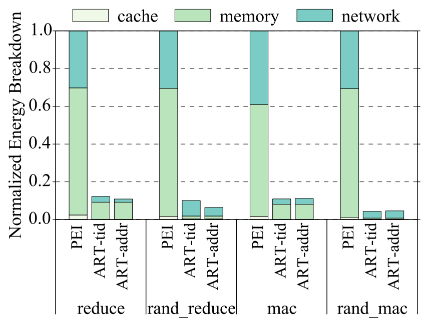


(b) Microbenchmarks

Fig. 14: Normalized power consumption over PEI.

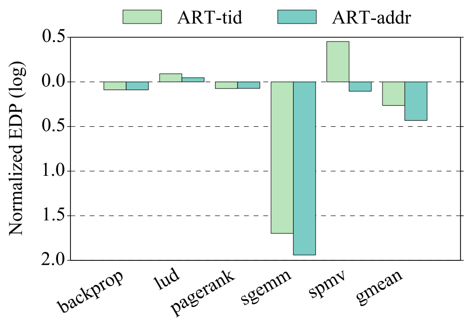


(a) Benchmarks

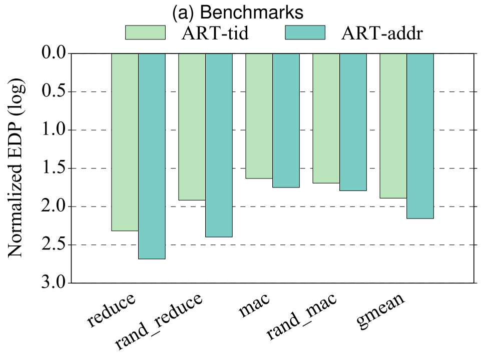


(b) Microbenchmarks

Fig. 15: Normalized energy consumption over PEI.



(a) Benchmarks



(b) Microbenchmarks

Fig. 16: Logarithmic scale of normalized energy-delay product (EDP) over PEI.

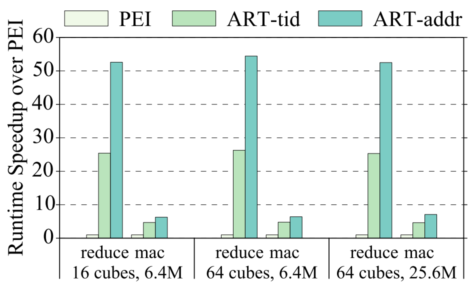


Fig. 17: Microbenchmarks on networks with 16 and 64 cubes with input vectors of 6.4 millions and 25.6 millions dimensions.

#### 6.2.2 Energy Consumption

Energy consumption is categorized into cache, memory and memory network, as shown in Fig. 15. ART-tid/addr reduces the energy consumption across all the benchmarks with regular memory access patterns and microbenchmarks. For applications that have irregular access patterns such as *pagerank*, the main contribution is from network energy due to the high overhead of fine-grained offloading. For *sgemm* and microbenchmarks, energy consumption is reduced dramatically owing to significant runtime speedup. We gain enormous benefit since most parts of these applications can be accelerated by *Active-Routing*.

#### 6.2.3 Energy-Delay Product

Fig. 16 shows the normalized energy-delay product (EDP) over PEI in logarithmic scale for energy efficiency. It shows that ART-tid/addr has lower EDP for all applications except for *spmv* with ART-tid. The runtime reductions and energy consumption jointly contribute to EDP reduction, achieving significant energy efficiency improvements. In *spmv* with ART-tid, the imbalanced work distribution leads to worse execution time. Since the energy saving is offset by the performance degradation, ART-tid has lower EDP on *spmv*. To summarize, ART-tid and ART-addr reduce the EDP by 80% on average compared to PEI.

### 6.3 Scalability

To evaluate scalability, we also experimented with *reduce* and *mac* on a 64-cube dragonfly memory network with the results shown in Fig. 17. For *mac* with the same problem size, ART-tid and ART-addr achieve 4.6× and 6.3× speedup compared to PEI on 16-cube memory network, whereas on 64-cube memory network, ART-tid and ART-addr outperform PEI for 4.7× and 6.4× improvements, respectively. As we scale the problem size four times as the memory capacity scales, ART-tid and ART-addr improve the performance of *mac* over PEI by 4.6× and 7.1×, respectively. Similarly, these techniques have the same trend on *reduce*. When comparing each technique's performance on the two different memory networks for the same problem size, PEI incurs 2% performance degradation on a 64-cube network compared to its performance on a 16-cube memory network. Whereas both ART-tid and ART-addr have less than 0.1% performance difference, either better or worse, on the two memory networks. Since PEI has more on/off chip data transfer than ART, it is more sensitive to the increased memory access latency due to higher average network latency in larger scale memory networks. In contrast, ART benefits from both network concurrency and memory parallelism, thereby scaling better for larger memory networks.

### 6.4 Vault-Level Parallelism and Offloading Granularity

Fig. 18 shows the runtime speedup of the original *Active-Routing* (ART-tid/addr), vault-level parallelism (VLP), page

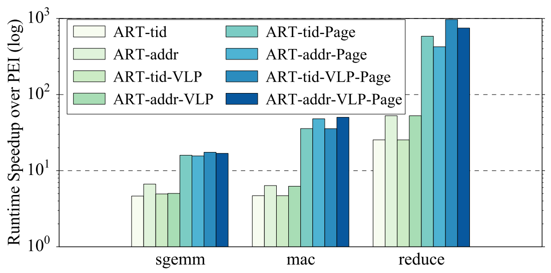


Fig. 18: Runtime speedup of original *Active-Routing* (ART-tid/addr), vault-level parallelism (VLP), page granular offloading (Page) and their combinations over PEI.

granular offloading (Page) and their combinations of *Active-Routing* normalized to PEI.

**VLP.** When applying VLP on top of cache block granular offloading, ART-tid-VLP and ART-addr-VLP show similar results as ART-tid and ART-addr. That is because the request arrival rates at memory cubes cannot fully utilize the compute throughput of the engine due to the offloading bottleneck created by the *many-to-few-to-many* offloading traffics from multicore CPU through a few memory controllers to many memory cubes. So ART-tid/addr is good enough to sustain the `Update` requests.

**Page.** To mitigate this bottleneck, we apply page granular offloading to pack more computation tasks in a single `Update` packet. ART-tid-Page and ART-addr-Page show that page granular offloading can improve the performance over cache granular offloading in ART-tid/addr by an order of magnitude in for both two-operand operations (*sgemm* and *mac*) and pure reduction (*reduce*). This indicates the cache block granular offloading fails to drive the peak computation throughput of *Active-Routing*, leaving the compute engine underutilized. It also shows that ART-tid-Page is better than ART-addr-Page on *reduce* because that the dramatically reduced offloading overhead mitigates the blocking at the memory ports, making ART-tid-Page more balanced for offloading compared to ART-addr-Page, which may have more `Updates` for some memory ports depending on memory access addresses.

**VLP-Page.** When applying VLP on top of page granular offloading, the performance is further improved by 5–10% for two-operand operations (*sgemm* and *mac*). For *reduce*, performance is further improved by 66% for ART-tid-VLP-Page and 76% for ART-addr-VLP-Page compared to ART-tid-Page and ART-addr-Page, respectively, which leads to three orders of magnitude speedup over PEI. It also implies that the page granular offloading alone is able to sustain the peak computation capability in the original *Active-Routing* (ART-tid/addr). Moreover, vault-level parallelism can improve the performance even further, especially for pure reduction as data is local to the compute vaults. Since two-operand operations need to have both operands from different vaults to be ready, even the offloaded computations arrive in bulk, such operand waiting time fails to extract the full VLP and achieves small improvement. Based on this characteristic, a cost-effective design can apply VLP only for pure reduction while keeping the two-operand computations in cube-level as in the original *Active-Routing*. In such a way, a design with one big ALU at cube-level for all computation supports and tiny ALUs at vault-level for pure reduction is sufficient to for performance gains, and moreover, it can save area cost and power with big and tiny ALUs instead of homogeneous big ALUs.

## 6.5 Dynamic Offloading: A Case Study

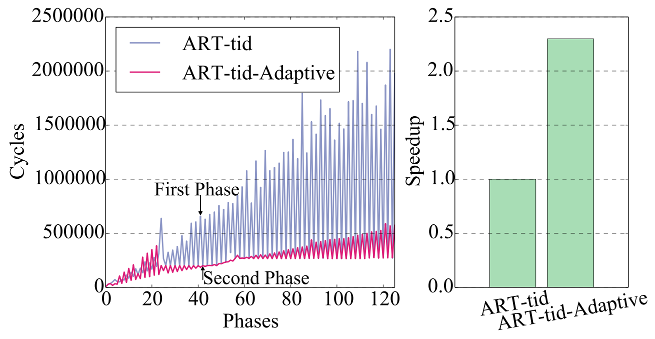


Fig. 19: LUD phase analysis and dynamic offloading.

Through an example, we show that further performance improvement can be achieved with a runtime knob that dynamically decides whether to offload computations (`Updates`) based on the memory access and communications patterns. Execution phases that exhibit good locality of data accesses experience performance benefits by exploiting cache hits when scheduled on the host CPU. In *lud*, it decomposes a matrix into upper and lower triangular matrices. Computations for these two matrices can be separated into two different phases, which execute alternatively with many iterations. The first phase is to compute the upper triangular matrix and the other to compute the lower triangular matrix. These two phases have different locality of data accesses. The second phase has a good data locality since it accesses the matrix in row-major order, whereas the first phase accesses the matrix in column-major order, showing poor locality.

For such a program behavior, the best execution model is to use *Active-Routing* for the first phase and process the second phase in the host processor. We analyze *lud*'s phase behaviors as shown in Fig. 19. In the ARTtid that always offloads computations to memory regardless of data locality, the number of cycles for first and second phases in each iteration dramatically increases and decreases. However, when we run ARTtid-adaptive in which computations of the first phase are offloaded to the memory and that of the second phase is processed in the host processor, we achieve 2× speedup.

## 7 Additional Related Work

**Near-Data Processing**. Recently, NDP architecture has become an active research area in computer architecture [9], [10], [12], [31]. Ahn et al. proposed a programmable PIM accelerator for large-scale graph processing [9]. More recently, Fujiki et al. [32] propose a programmable in-memory processor architecture, and data-parallel programming framework using non-volatile memory. Mondrian [11] takes an algorithm-hardware co-design approach to sequence irregular accesses for better locality by pre-processing. Recent study [33] discovered the data movement as the bottleneck for performance and energy efficiency by analyzing

Google workloads. While it focuses on consumer devices and *Active-Routing* target for high-performance processors. Most recently, domain-specific PIM architectures have been designed for different applications, ranging from deep learning [34], [35], to image processing [36], to graph processing [37]. The coherence mechanism designed for near-data architectures can also be applied to *Active-Routing* to reduce the unnecessary coherence traffic [22].

**In-Network Computing**. Previous research [14], [15], [16] has advocated interconnection networks to offer more functionalities on top of normal switching purposes. Active Message [14] embeds the function pointer and arguments across the network to perform tasks in remote compute nodes. Pfister et al. [15] and Ma [38] proposed mechanisms to combine messages so as to reduce network traffic. Recently, IncBricks [16] implements an in-network caching middlebox for key-value acceleration in router switches. Several studies [17], [18], [19] proposed mechanisms to optimize shared value update or reduction in the network. Although these mechanisms support data processing in the network but still suffer the burden of data movement from memory to CPU, while *Active-Routing* tackles this issue. Recently, Li et al. proposed iSwitch to accelerate reduction for distributed reinforcement learning training in parameter server [39]. Similar to *Active-Routing*, an in-network reduction architecture has been developed to accelerate collective reduction in multiprocessor shared memory [40]. Different from these work, *Active-Routing* applies to near-data processing paradigm and also optimizes aggregation of intermediate results of arithmetic operators, which is not supported in these proposals.

## 8 Conclusions

We propose *Active-Routing*, an in-network computing architecture, to enable reduction en-route in data-intensive applications for near-data processing. *Active-Routing* is implemented as a novel three-phase processing procedure, which offloads the computation near data in the memory network for execution and aggregates the results along their routing path. We categorize memory access patterns of compute kernels of interest and propose various offloading granularity by exploiting data locality to reduce offloading overhead. We further extend the original *Active-Routing* to enable vault-level parallelism for computation throughput improvement. Compared to the state-of-the-art PIM architecture, *Active-Routing* can achieve up to 7× speedup with a geometric mean of 60% performance improvement and reduce energy-delay product by 80% on average across benchmarks. Aggressive parallelism and offloading optimizations show further improvements of an order of magnitude, showing promising potential for in-network computing and data-flow processing.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback. This work was done while Jiayi Huang, Troy Fulton, and Ramprakash Reddy Puli were with Texas A&M University.

## References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *International Conference on Neural Information Processing Systems (NIPS)*. Curran Associates Inc., 2012, pp. 1097–1105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257

[2] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores," in *International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 2015, pp. 44–55. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2015.11

[3] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Hot Chips 23 Symposium (HCS)*. IEEE, 2011, pp. 1–24.

[4] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin *et al.*, "25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*. IEEE, 2014, pp. 432–433.

[5] G. H. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors," in *International Symposium on Computer Architecture (ISCA)*, 2008, pp. 453–464.

[6] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-Centric System Interconnect Design with Hybrid Memory Cubes," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Press, 2013, pp. 145–156.

[7] J. Zhan, I. Akgun, J. Zhao, A. Davis, P. Faraboschi, Y. Wang, and Y. Xie, "A Unified Memory Network Architecture for In-Memory Computing in Commodity Servers," in *International Sympoium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–14.

[8] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute Caches," in *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*, 2017, pp. 481–492.

[9] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 105–117.

[10] B. Hong, G. Kim, J. H. Ahn, Y. Kwon, H. Kim, and J. Kim, "Accelerating Linked-List Traversal through Near-Data Processing," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2016, pp. 113–124.

[11] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian Data Engine," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17, 2017, pp. 639–651.

[12] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 336–348.

[13] J. Ahn, S. Yoo, and K. Choi, "AIM: Energy-Efficient Aggregation inside the Memory Hierarchy," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, p. 34, 2016.

[14] T. V. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *International Symposium on Computer Architecture (ISCA)*. IEEE, 1992, pp. 256–266.

[15] G. F. Pfister and V. A. Norton, ""Hot Spot" Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers*, vol. c-34, no. 10, pp. 943–948, 1985.

[16] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "IncBricks: Toward In-Network Computation with an In-Network Cache," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017, pp. 795–809. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037731

[17] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, vol. c-32, no. 2, pp. 175–189, 1983.

[18] D. K. Panda, "Global Reduction in Wormhole $k$-ary $n$-cube Networks with Multidestination Exchange Worms," in *International Parallel Processing Symposium (IPPS)*, 1995, pp. 652–659.

[19] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q Interconnection Network

and Message Unit," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–10.

[20] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 461–475.

[21] J. Huang, R. R. Puli, P. Majumder, S. Kim, R. Boyapati, K. H. Yum, and E. J. Kim, "Active-Routing: Compute on the Way for Near-Data Processing," in *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA-25)*, February 2019, pp. 674–686.

[22] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng *et al.*, "CoNDA: Efficient cache coherence support for near-data accelerators," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 629–642.

[23] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A Manycore Simulator with Application-Level+ Simulation and Detailed Microarchitecture Modeling," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 74–85.

[24] D. I. Jeon and K. S. Chung, "CasHMC: A Cycle-Accurate Simulator for Hybrid Memory Cube," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 10–13, Jan 2017.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[26] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *International Symposium on Microarchitecture (MICRO)*, 2007, pp. 3–14. [Online]. Available: https://doi.org/10.1109/MICRO.2007.30

[27] M. Poremba, I. Akgun, J. Yin, O. Kayiran, Y. Xie, and G. H. Loh, "There and Back Again: Optimizing the Interconnect in Networks of Memory Cubes," in *International Symposium on Computer Architecture (ISCA)*. ACM, 2017, pp. 678–690. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080251

[28] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2010.5650274

[29] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," Available from http://snap.stanford.edu/data, Jun. 2014.

[30] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois at Urbana-Champaign, Tech. Rep., March 2012.

[31] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *International Symposium on Microarchitecture (MICRO)*. ACM, 2017, pp. 273–287. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124544

[32] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 1–14.

[33] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan *et al.*, "Google workloads for consumer devices: mitigating data movement bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 316–331.

[34] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 790–803.

[35] W. Li, P. Xu, Y. Zhao, H. Li, Y. Xie, and Y. Lin, "TIMELY: Pushing Data Movements and Interfaces in PIM Accelerators Towards Local and in Time Domain," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 832–845.

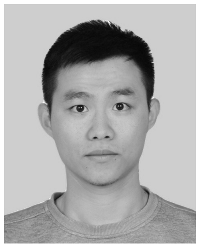
[36] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie, "iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 804–817.

[37] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "GraphQ: Scalable PIM-based Graph Processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.

[38] S. Ma, N. E. Jerger, and Z. Wang, "Supporting Efficient Collective Communication in NoCs," in *High Performance Computer Architecture (HPCA)*. IEEE, 2012, pp. 1–12.

[39] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating Distributed Reinforcement Learning with In-Switch Computing," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 279–291.

[40] B. Klenk, N. Jiang, G. Thorson, and L. Dennison, "An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 996–1009.



**Jiayi Huang** (Member, IEEE) received the BEng degree in information and communication engineering from Zhejiang University, China, in 2014, and the PhD degree in computer engineering from Texas A&M University, in 2020. He is currently a postdoctoral researcher with the Department of Electrical and Computer Engineering, UC Santa Barbara. His research interests include computer architecture and systems, with a special focus on communication/data-centric and heterogeneous architecture. He is a member of ACM and IEEE Computer Society.

**Pritam Majumder** received the BTech degree in computer science and engineering from WBUT India, in 2011, and the MS degree in computer science and engineering from Indian Institute of Technology, Madras, in 2015. He is working toward the Ph.D. degree with the Department of Computer Science and Engineering, Texas A&M University. His research interests include the fields of computer architecture, systems, and machine learning. He is a student member of ACM.

**Sungkeun Kim** received the BEng degree in computer science and engineering from Kyungpook National University, Republic of Korea, in 2011. He is working toward the PhD degree with the Department of Computer Science and Engineering, Texas A&M University. His research interests include computer architecture and systems, focusing on networks-on-chip, memory systems and near-data processing, and hardware security. Before starting a Ph.D., he worked as a software engineer at Samsung Electronics, Suwon, Republic Korea.

**Troy Fulton** received the BS degree in computer science from Texas A&M University, in 2020, where he participated with the Undergraduate Research Scholars Program and completed his thesis on parallelizing In-Memory Computations for Active-Routing. He currently works as a software engineer at Aspen Insights, LLC, a company focused on applying artificial intelligence to improve clinical research in healthcare. His research interests include computer architecture and compiler design.

**Ramprakash Reddy Puli** received the BTech degree in electrical engineering from Indian Institute of Technology, Kharagpur, India, in 2014, and the MS degree in computer engineering from Texas A&M University, in 2018. He currently works as a Sr. Architect at Nvidia to enable next generation high bandwidth memory architectures for GPUs. His research interests include computer architecture, high performance computing, and DRAM technologies.

**Ki Hwan Yum** (Member, IEEE) received the BS degree in mathematics from Seoul National University, Korea, in 1989, the MS degree in computer science and engineering from Pohang University of Science and Technology, Korea, in 1994, and the PhD degree in computer science and engineering from the Pennsylvania State University in 2002. From 1994 to 1997, he was a member of Technical Staff in Korea Telecom Research and Development Group. He is currently a research assistant professor in the Department of Computer Science and Engineering, Texas A&M University. His research interests include computer architecture, parallel/distributed systems, cluster computing, and performance evaluation. He is a member of IEEE Computer Society and ACM.

**Eun Jung Kim** (Member, IEEE) received the BS degree in computer science from KAIST, Korea, the MS degree in computer science from Pohang University of Science and Technology, Korea, and the PhD degree from the Department of Computer Science and Engineering, Pennsylvania State University. She is an associate professor with the Department of Computer Science and Engineering, Texas A&M University. Her research interests include computer architecture, power efficient systems, parallel/distributed systems, cluster computing, security, and sensor network. She worked as a member of technical staff in Korea Telecom for three years. She is a member of IEEE Computer Society. More information about her research is available at http://faculty.cse.tamu.edu/ejkim.