Active-Routing: Compute on the Way for Near-Data Processing

Jiayi Huang[†], Ramprakash Reddy Puli[‡], Pritam Majumder[†], Sungkeun Kim[†], Rahul Boyapati[§]
Ki Hwan Yum[†] and Eun Jung Kim[†]
[†]*Texas A&M University,* [‡]*NVIDIA,* [§]*Intel Corporation*
{*jyhuang,pritam2309,ksungkeun84,yum,ejkim*}*@cse.tamu.edu, rpuli@nvidia.com, rahul.boyapati@intel.com*

***Abstract*—The explosion of data availability and the demand for faster data analysis have led to the emergence of applications exhibiting large memory footprint and low data reuse rate. These workloads, ranging from neural networks to graph processing, expose compute kernels that operate over myriads of data. Significant data movement requirements of these kernels impose heavy stress on modern memory subsystems and communication fabrics. To mitigate the worsening gap between high CPU computation density and deficient memory bandwidth, solutions like memory networks and near-data processing designs are being architected to improve system performance substantially.**

In this work, we examine the idea of mapping compute kernels to the memory network so as to leverage in-network computing in data-flow style, by means of near-data processing. We propose *Active-Routing*, an in-network compute architecture that enables computation on the way for near-data processing by exploiting patterns of aggregation over intermediate results of arithmetic operators. The proposed architecture leverages the massive memory-level parallelism and network concurrency to optimize the aggregation operations along a dynamically built *Active-Routing Tree*. Our evaluations show that *Active-Routing* can achieve upto 7× speedup with an average of 60% performance improvement, and reduce the energy-delay product by 80% across various benchmarks compared to the state-of-the-art processing-in-memory architecture.

***Keywords*-memory network; data-flow; in-network computing; near-data processing; processing-in-memory**

I. Introduction

With the improvement of technology and advent of numerous network connected devices, the amount of data generated has been exploding. This leads to an increasing demand for fast data analysis to extract values from these humongous amounts of data. Hence, data analytic applications that process these bulk of data exhibit large memory footprint and low data reuse rate. These workloads, ranging from neural networks to graph processing [1], [2], have simple compute kernels that operate over a myriad of data. The simple computations of these kernels and the large amounts of data to be processed cause significant data movements across the memory hierarchy. As a result, modern memory subsystems and communication fabrics are under enormous pressure. Furthermore, due to the gap between dense CPU computation power and deficient data supply, computer systems fail to achieve their peak computational capability. Therefore, architectural innovations are imperative to reduce data movement for gaining substantial improvements in terms of system performance as well as energy efficiency.

Recently, a significant amount of research efforts have been made for designing data-centric computer systems. To keep pace with processors' computation capabilities, new memory designs such as Hybrid Memory Cube (HMC) [3] and High Bandwidth Memory (HBM) [4] provide higher bandwidth by utilizing 3D stacking [5]. In addition, the traditional processor-centric design is not cost-effective to scale memory capacity and is suboptimal for system bandwidth provision [6]. On the other hand, memory-centric designs are proposed to connect memory modules to form a memory network as a large memory pool as well as to fully utilize processor and memory bandwidth [6], [7]. These design adoptions may alleviate the data response bottleneck, but still require a considerable amount of data movement due to imposing heavy stress on the communication fabrics and consuming excessive energy.

Previous research has proposed various techniques to reduce data movement across the memory hierarchy to improve the system efficiency. Near-data processing (NDP), as a promising compute paradigm, has driven new architectures to move computations near data-resident locations, such as cache and memory. Aga et al. proposed compute cache [8] that uses bit-line circuit technology to perform simple computation in the cache to enable in-place computing. Processing-in-memory (PIM) [9], [10], [11], [12], [13], [14], [15], [16], [17] is an alternative NDP design that introduces compute elements in memory for data processing. Recent studies [18], [19] have proposed to integrate PIM architectures within modern systems in a seamless fashion. They extended the instruction set to offload computations to data-resident memory modules. These mechanisms achieve better efficiency compared to conventional computing due to reduced data movements. They are most effective in the case of irregular memory accesses and atomic write operations. However, they are suboptimal when performing simple tasks over a large size of raw data, such as *dot product*, since they need to fetch part of the data across the memory network for further processing when data are not located in the same module that incurs communication and energy overhead.

Prior research [20], [21], [22] has advocated to provide computation power as well as routing functionalities in communication fabrics. The NYU Ultracomputer [23] introduced adders in routers to combine fetch-and-update requests for the same shared variable. Panda [24] and Chen et al. [25] proposed similar hardware to optimize reduction in the network interface for MPI collective communications. These mechanisms only support pure reduction operations and cannot accelerate operations like *dot product*, thus require significant data movements across the memory hierarchy to first compute the intermediate results. Recently, Kwon et al. proposed MAERI [26] to improve efficiency for data-flow computations in deep neural network accelerators, which does not target general applications. The multiply operations require data to be brought to local SRAM and are calculated only at leaf nodes in the tree-based network topology. These in-network compute solutions have limited adaptivity since the reduction tree/ring is statically tied to the network topology. As part of our proposal, we redesign the interconnect fabric to support more diverse compute operations and to provide *topology-oblivious* dynamic routing tree for reduction acceleration.

We propose *Active-Routing*, an in-network compute architecture that enables compute on the way for near-data processing. We examine the idea of mapping compute kernels to the memory network for data-flow style processing by exploiting the pattern of aggregation over intermediate results of arithmetic operators. It seeks to schedule computations at routers attached to memory so as to take advantage of the massive bandwidth and parallelism in memory. Meanwhile, it dynamically builds topology-oblivious *Active-Routing trees* and leverages the network concurrency to optimize reduction operations along the routing path. We also categorize the memory access patterns of the operands for arithmetic operators and propose optimizations to exploit both regular and irregular memory accesses.

The major contributions of this paper are as follows.

- We propose an in-network compute architecture, *Active-Routing*, which moves computation closer to data in the memory network, and aggregate compute results on the way along the routing path.
- We present a novel mechanism with three-phase packet processing: *Active-Routing Tree* construction, *Update Phase* for data processing, and *Gather Phase* for *Active-Routing* reduction. It dynamically builds *topology-oblivious* routing trees to optimize reduction operation following data processing by exploiting massive memory-level parallelism and network concurrency.
- We categorize memory access patterns of processed data into three groups, and propose enhancement techniques with various offloading granularities to amortize offloading overhead by leveraging their characteristics.
- Our evaluations show that *Active-Routing* can achieve up to 7$\times$ speedup with an average of 60% performance improvement and reduce energy-delay product by 80% on average across various benchmarks over the state-of-the-art processing-in-memory architecture.

The rest of the paper is organized as follows. Section II presents the background for this research. In Section III we introduce *Active-Routing* followed by Section IV describing its implementation. We present the evaluation methodology in Section V and analyze our experiment results in Section VI. The related work is detailed in Section VII. Finally, we conclude our work in Section VIII.

II. Background

In this section, we first introduce die-stacked memory and its support for processing-in-memory (PIM). Then we explain the memory network and the potential of in-network computing for enhancing near-data processing.

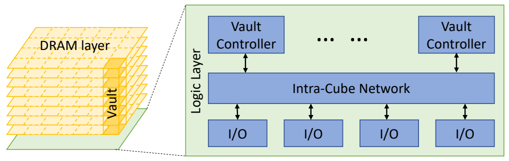


Figure 1. Hybrid Memory Cube

A. Die-Stacked Memory

Advancements in memory technology have facilitated the integration of logic and memory dies using 3D stacking [5]. In die-stacked memory, DRAM layers are stacked on top of a logic layer. The DRAM layers are connected with the logic layer using high bandwidth, and low-latency Through-Silicon Vias (TSV). Hybrid Memory Cube (HMC) [3] and High Bandwidth Memory [4] are two popular examples of die-stacked memory. Without loss of generality, we demonstrate *Active-Routing* using HMC in this paper. Note that it can also be applied to other memory technology and interconnects like HBM and active interposers. Figure 1 shows the organization of HMC, which is partitioned vertically into several vaults consisting of multiple TSV connections to the logic layer. Each vault is controlled by a vault controller implemented on the logic die. The vault controllers are sparsely placed and that leaves ample amount of unused silicon area to deploy more complex functional logics. It has been used to implement computation capability ranging from limited functionality [18], [12], [11] to full-fledged processors [9], [15]. HMC communicates with processor or other memory cubes through four ports. Inside the cube's logic die, an intra-cube network is used to route the packets between the vaults and ports. HMC also enables larger memory size per package and provides abundant internal and external bandwidth with TSVs and high-speed link protocol. These advantages are leveraged in many existing PIM studies [9], [15].

B. Memory Network

Conventional systems with DDR memory have capacity limits and bandwidth bottleneck due to the limited number

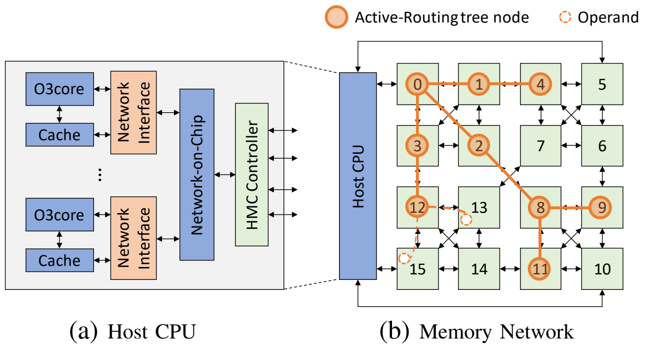


Figure 2. System Configuration with (a) a Host CPU Connected to (b) a Memory Network with an ***Active-Routing*** example.

of pins per processor chip. Therefore, it requires more processor sockets in such systems to scale their memory capacity. However, the overweight data movement with respect to light computation in emerging data-centric applications can lead CPU to be under-utilized. In contrast, HMCs can be chained together to form a cost-effective memory network using packet switching and provide large memory capacity. In addition, commonly adopted processor-centric design optimizes processor-to-processor communication but overlooks the overall system bandwidth utilization. A recent study [6] has shown that memory-centric designs can achieve better bandwidth utilization as compared to processor-centric designs.

C. In-Network Compute Potential

Supported by the advanced die-stacked memory technology, PIM architectures have been widely studied to realize near-data processing. With high-bandwidth and large memory capacity provision, a memory network is adopted to scale PIM architectures for accelerating data-centric applications [9]. One unsolved problem of such a system is how to determine the compute point for data located remotely. This computation can be mapped to a data-flow graph and naturally scheduled as network flows along with computation. In-network computing, which takes communication into account, can further reduce data movement and improve system efficiency by exploiting memory-level parallelism as well as network concurrency. In this paper, we propose *Active-Routing* as a step towards in-network computing.

III. Active-Routing Architecture

In this section, we first illustrate *Active-Routing* by walking through an example. Then we describe its three-phase packet processing procedure. Lastly, we categorize the memory access patterns of the data to be processed and propose enhancements to reduce offloading overhead by leveraging their characteristics.

A. Architectural Overview

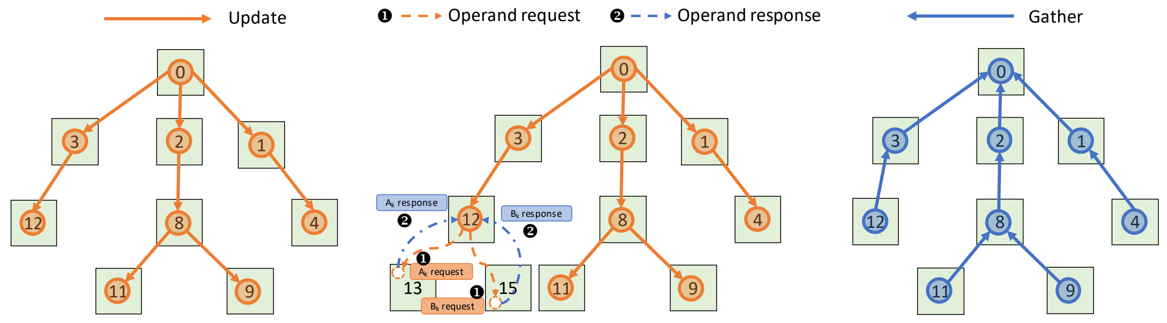
Figure 2 presents the system configuration, where host processors are connected to a memory network formed by chaining HMCs. In this system, we show an example of *Active-Routing* in the memory network that computes `sum += A[i]×B[i]` over a large loop with loop-index `i`. Each computation of `A[i]×B[i]` is offloaded from host CPU to memory network as an **Update** packet. **Update** packets are scheduled for computation at the memory cubes near to the operand locations to compute the partial sum through NDP. After **Update** offloading, a **Gather** packet is sent to collect the partial results from each cube, and reduce them in the network routers on the way back to the host.

Figure 3 shows the three phases of *Active-Routing* as it progresses in the timeline for this example, namely *ARTree Construction*, *Update* and *Gather Phase*.

- While offloading **Update** packets, an *Active-Routing Tree (ARTree)* is being constructed along the packets' paths to the scheduled compute memory cubes. For example, in Figure 2 (b), an **Update** packet is sent from CPU through memory cube 0 to cube 8. It records the tree nodes and builds a tree branch along its path to cube 8. **Update** packets scheduled at different cubes construct different branches. These branches altogether form an *ARTree*, as abstracted in Figure 3 (a).
- The offloaded computations drive near-data processing during the *Update Phase* as shown in Figure 3 (b). Each operation `A[i]×B[i]` needs to request its source operands `A[i]` and `B[i]` to finish the computation and update the partial sum in the scheduled cube. Figure 3 (b) also shows a case where two operands do not reside in the same cube. In such scenarios, the **Update** packet will be sent to the scheduled compute point that is the last common cube of the minimum routes (cube 12) for both operands: ❶ it replicates to issue two operand requests for $\mathtt{A}_k$ and $\mathtt{B}_k$ to the resident memory cube 13 and cube 15, respectively. ❷ Then, the two operand responses are replied to cube 12 to complete the computation. All the intermediate results in the same compute cube are reduced to a partial sum in the cube during this phase.
- Figure 3 (c) shows the *Gather Phase* when **Gather** packet is issued after sending all the **Update** packets. It is replicated from the root to each node of the *ARTree*. Then **Gathers** at leaf nodes initiate network reduction of partial sums computed in the previous phase in dataflow manner to the root along the *ARTree*.

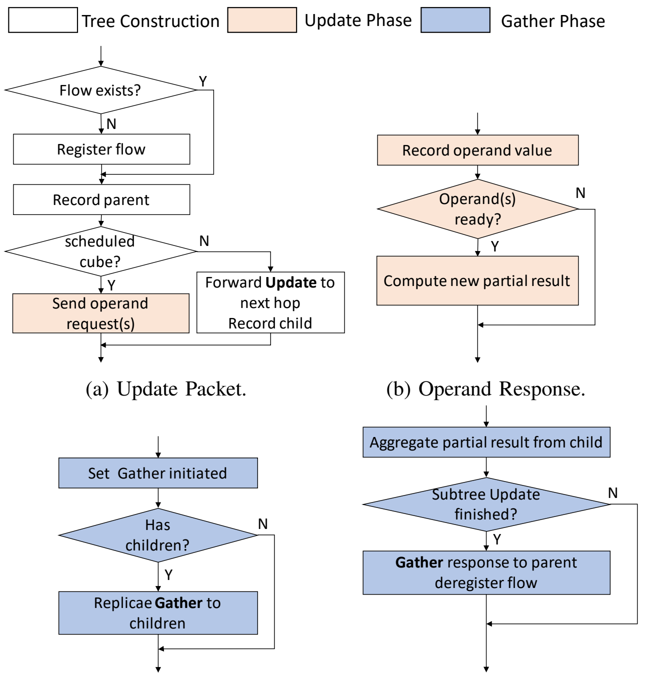
B. Three-Phase Packet Processing

In general, *Active-Routing* maps a compute kernel in memory network to optimize reduction over intermediate results of arithmetic operators. We name such a mapping as an *Active-Routing flow*. A unique identification (*flow ID*) is



(a) Active-Routing Tree Construction. (b) Update Phase. (c) Gather Phase.

Figure 3. Active-Routing Consists of Three Phases: (a) Active-Routing Tree Construction on-the-fly; (b) Near-Data Processing in Update Phase; and (c) Network Aggregation along the Active-Routing Tree in Gather Phase.



(a) Update Packet. (b) Operand Response. (c) Gather Request. (d) Gather Response.

Figure 4. Active Packet Processing Flow Chart for (a) Update Packet, (b) Operand Response Packet, (c) Gather Request Packet and (d) Gather Response Packet.

assigned for each *flow* and its corresponding *ARTree*. Each *flow* involves a three-phase packet processing procedure as shown in Figure 4.

ARTree Construction. For each *flow*, an *ARTree* is built dynamically while processing its **Update** packets, as shown in Figure 4 (a). Upon receiving an **Update** packet, each cube registers its *flow ID*. If the **Update** packet is not scheduled to compute at the current cube, the packet is forwarded to its child based on its routing to the scheduled compute cube. Therefore, an *ARTree* is built by recording parent and children information at each node.

Update Phase. This phase starts in parallel with the *ARTree* construction phase. It involves processing of **Update** packets, and operand request/response packets as shown in Figures 4 (a) and (b). While processing **Update** packets, operand requests are sent out to the memory from the scheduled compute node. When the operand responses arrive, the arithmetic operations are scheduled to compute the partial aggregation result.

Gather Phase. Figures 4 (c) and (d) show the packet processing in *Gather Phase* to commit *Active-Routing flow*. This phase has one forward pass to spread the **Gather** requests from the root to leaf nodes, and a backward pass to reduce the results from leaf nodes to the root node. Once a node's subtree finishes *Update Phase*, it replies to its parent and deallocate the *flow* record. A parent receives **Gather** responses from all its children to indicate the completion of their *Update Phase*. When the root node finishes its own *Update Phase* and receives all its children's **Gather** responses, it commits the *flow*.

C. Memory Access Patterns

Instruction offloading and operand fetching incur overhead using packet switching due to metadata in the packet header and packet internal fragmentation. Memory access patterns of operand fetching can be exploited to amortize the overhead by offloading multiple operations at a time. *Active-Routing* aims to optimize reduction on massive intermediate results of arithmetic operators, such as $\texttt{sum} = \sum_{i=1}^{n} \star\texttt{A}_\texttt{i} \times \star\texttt{B}_\texttt{i}$, where $\texttt{A}_\texttt{i}$ and $\texttt{B}_\texttt{i}$ store the operand addresses. Memory access patterns of operands can be regular when vector `A` stores array addresses. When it stores addresses of graph nodes or sparse matrix elements, the access pattern tends to be irregular. Therefore, the combined memory access pattern for the two operands can be categorized into three groups: *regular-regular*, *regular-irregular*, and *irregular-irregular*. Based on these three categories, we propose three different ways to leverage data locality.

For *regular-regular* access pattern, we offload the computation in cache block granularity as vector processing. While for *regular-irregular* access pattern, we fetch the irregular data and send them to the regular data resident location for processing. The above two methods maximize the locality benefit and reduce the memory accesses. For *irregular-irregular* memory access pattern, we simply fetch single operand pairs to the scheduled compute node as scalar operations. *Active-Routing* can cooperate with previous study [15] to further optimize *irregular-irregular* access pattern, which we leave for future work.

```
// baseline implementation
global diff = 0.0;
local loc_diff = 0.0;
for (v: v_start to v_end) {
  loc_diff += abs(v.next_pagerank - v.pagerank);
  v.pagerank = v.next_pagerank;
  v.next_pagerank = 0.15 / graph.num_vertices;
}
atomic diff += loc_diff;


// active optimization
global diff = 0.0;
local temp = 0.15 / graph.num_vertices;
for (v: v_start to v_end) {
  Update(&v.next_pagerank, &v.pagerank, &diff, abs);
  Update(&v.next_pagerank, nil, &v.pagerank, mov);
  Update(temp, nil, &v.next_pagerank, const_assign);
}
Gather(&diff, num_threads);
```

Figure 5. Pseudocode of Thread Worker for Parallel PageRank.

IV. Implementation

In this section, we describe the programming interface and instruction set architecture (ISA) extension. Then we introduce the hardware components that work in synergy to realize *Active-Routing*, including Network Interface (NI) support and *Active-Routing Engine (ARE)*. Lastly, we discuss system integrity considerations and several enhancements in *Active-Routing*.

A. Programming Interface and ISA Extension

We provide simple programming interfaces (**Update** and **Gather**) to translate the program semantics into extended instructions. The ISA extensions are used to communicate with Network Interface to offload computations to memory network for *Active-Routing* processing.

```
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);
Gather(void *target, int num_threads);
```

The above **Update** and **Gather** APIs are defined to offload *Active-Routing flows*. The **Update** API carries two source memory addresses of an arithmetic operation. The postfix **RR**, **RI** and **II** of **Update** API are used for three memory access pattern categories, respectively. The `op` parameter is the opcode indicating the type of arithmetic and reduction operation (e.g. floating point multiply-and-accumulate). The `target` field in both APIs is the address of the reduced variable, which is hashed to a unique identification for each *flow*. In **Gather** API, the `num_threads` parameter indicates the number of threads working on the *flow*. It is used for an implicit barrier at the root of *ARTree* to guarantee all the **Updates** have been initiated. We generalize the **Update** API with opcode `op` to support simple operations such as assignment. These APIs are translated to extended instructions by the compiler. During execution, instruction fields are written to a set of dedicated registers in the Network Interface (NI). NI can assemble this information into an **Update** or a **Gather** packet and send it to the memory network.

Figure 5 shows the baseline and *Active-Routing* implementations of the thread worker pseudocode of *pagerank* calculation kernel. In the baseline implementation, the **atomic** update for `diff` needs to fetch the `pagerank` and `next_pagerank` values for each vertex in the graph. This consumes a large amount of bandwidth due to irregular graph access patterns. It also needs to reduce the `diff` value atomically for each thread, which causes high overhead and limits thread scaling. In contrast, *Active-Routing* allows updates of `diff` near the data location to save bandwidth. In addition, the **Gather** requests from all the threads of same *flow* are synchronized at the root of *ARTree* as an implicit barrier. Then reduction is initiated along the *ARTree*. Note that the read-write dependencies between instructions are enforced as same as normal instructions. The read-write dependencies can be tracked and resolved by memory controllers similar to read-write requests dependencies handling with simple extension.

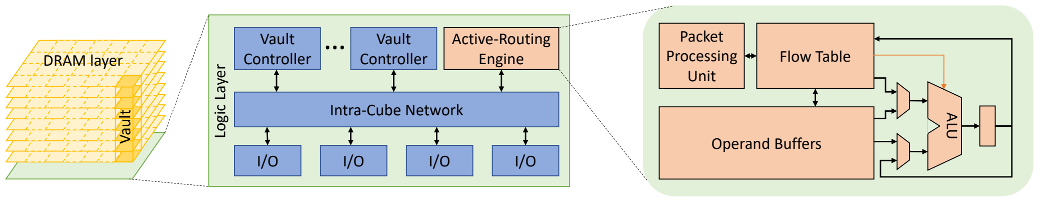
B. Network Interface

Programming interfaces are used in application for *Active-Routing* offloading. Compiler takes the API and translates it into extended instructions. Extended instructions are assembled to packets and offloaded to the memory network for processing. This functionality can be added to Network Interface (NI), connecting core and on-chip network, with marginal change. In NI, we add dedicated registers that can be written by extended instructions to convey the opcode and operand information. NI reads these registers to assemble an **Update** or a **Gather** packet and issue it into the memory network.

C. Active-Routing Engine

The *Active-Routing* functionalities are implemented in *Active-Routing Engine (ARE)* on the HMC logic layer as shown in Figure 6 (a). It is integrated as an attached module 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* to store operands, 4) an ALU for computation.

1) Packet Processing Unit: Packet processing unit is responsible for decoding the **Update** and **Gather** packets and schedule actions correspondingly as shown in Figure 4. It



(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.

Figure 6. Active-Routing Microarchitecture: (a) Engine Implementation in HMC Logic Layer with (b) Flow Table Entry and (c) Operand Buffer Entry.

can generate operand request packets to fetch the data and **Gather** response to commit the partial result to its parent.

2) Flow Table: Flow table keeps track of both the structure and states information of each *flow*. Figure 6 (b) shows a *flow* entry. Each entry in the table is a tree node record that maintains the structure of the tree by keeping a unique `flow ID`, an `opcode` for computation, and its `parent` and `children`. It also keeps the *flow*'s state, including the partial `result`, the `req_count` and `rep_count`, as well as `Gflag`. The `req_count` and `rep_count` counters are used to keep the number of issued requests and committed operations. A *Update Phase* is considered finished when these two counters are the same. The `Gflag` is set by a **Gather** request indicating that the flow can start reduction once *Update Phase* completes.

3) Operand Buffers: **Update** packets are processed to generate request(s) to fetch operands and perform the computation with the response. Operand buffer is used as a temporary storage for the operands waiting to be processed, therefore maintaining the pending **Update** operations. We make a pool of operand buffers shared by different flows so as to improve the throughput and reduce the overhead. An operand buffer entry is reserved before sending out operand request(s) since co-existing *flows* can easily cause deadlock due to wait-and-hold condition especially for two-operand operations. Figure 6 (c) shows an operand buffer entry, which keeps the `flowID` and `opcode`, two `operand` fields and two `ready` flags to indicate the operand's availability. To reduce the operand buffer access time, we use a free and a ready queue to keep IDs of free and ready operand entries, respectively, for ease of direct lookup.

4) ALU: A light-weight ALU is implemented in ARE to compute arithmetic operations. *Active-Routing* supports various operations on different data types, including reduction operations such as sum, xor, and, min, and max, as well as multiply-accumulate on floating point and integer data. We plan to generalize our approach and implement more powerful logics to support complex program accelerations.

5) Putting It All Together: Upon receiving an **Update** request packet, ARE processes it in the Packet Processing Unit. If the corresponding *flow* has not yet registered in the *flow table*, an entry is allocated for the new *flow*. The flow is registered and fields are initialized by recording 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 computed route to next hop, 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 embedding the operand address and buffer entry ID are also generated. If all operand buffer entries are busy, the packet processing unit is stalled until an operand buffer entry is available. When a response for the operands arrives, the corresponding operand buffer entry is updated. If operands are ready, the operand entry ID is pushed to the ready queue for processing. ALU is directed by the ready queue for computation. After the computation finishes, the `resp_count` is incremented and `result` is updated in the corresponding *flow* entry. The operand buffer is deallocated for reuse by pushing back its ID to free queue. 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 every time the `result` is updated by either computation in current cube or **Gather** packet from a child, if `Gflag` is set and `children flags` are cleared, a **Gather** packet is generated to send the partial result back to its parent and release the flow table entry.

D. Integrity Considerations

There are two important design considerations to seamlessly integrate *Active-Routing* into current computer systems: (1) virtual memory support and (2) cache coherence.

1) Virtual Memory: Since *Active-Routing* is implemented by ISA extension, 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. With this design principle, we can avoid overhead for address translation units in the directories, or memory.

2) Cache Coherence: To offload instructions for *Active-Routing* optimization, it should ensure that the offloaded *flow* is using the up-to-date data in memory. A naïve way is allocating uncacheable memory for the data that may be used in the optimization. However, it may hurt the performance in other program execution phases which can use the deep cache hierarchy to exploit locality. To work around with coherence, offloaded packets are first sent to the directory based on their address, and query for back-invalidation if data is cached on-chip similar to [18]. Then it will be issued to the memory for *Active-Routing* processing. Since **Update** packets are issued in parallel, the back-invalidation overhead is amortized across massive concurrent packets. We observe that back-invalidation rarely happens in our experiments.

E. Enhancements in Active-Routing

We observe two critical points that have significant impact on the *Active-Routing* performance: (1) the decision for choosing the root of a tree affects the network congestion, and (2) the overhead of offloading computations widely varies with the change in its granularity. To improve *Active-Routing* performance further, we address each of these points as follows.

Since the computations are offloaded from host CPU through the memory ports, we naturally consider the cubes that are attached to the four channel ports as root node candidates. We start with a static approach that we always assign the root node to be cube 0. In order to balance the load better in the network, we propose two enhancement techniques that can consider all four corner cubes as roots and are able to create multiple trees for one *flow*. The first one uses thread ID to interleave the candidate cubes so as 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. 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 number of memory accesses, we adapt the offloading granularity, to exploit the data locality of different memory access patterns discussed in Section III-C. This optimization is applied to both ART-tid and ART-addr, whereas Naïve-ART does not consider granularity, which simply offloads every single operand pair

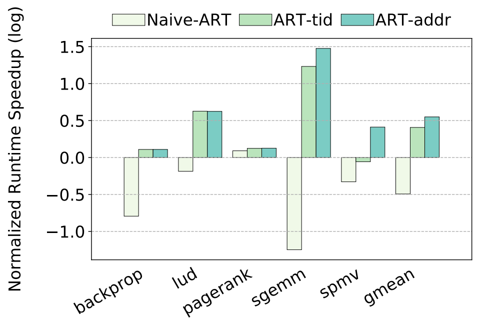


Figure 7. Runtime Speedup of ART Variants over HMC Baseline.

without considering data locality. 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.

Figure 7 shows the improvement impacts of enhancements over Naïve-ART. We take the log scale of speedup that is normalized to HMC conventional system baseline (not shown). It shows that with naïve way of static tree formation and offloading, Naïve-ART is worse than HMC baseline, especially when there is some locality in accesses. In contrast, by constructing the trees dynamically and exploiting the memory access patterns, we can achieve better performance. In the following sections, we only present ART-tid and ART-addr for detail analysis.

V. Methodology

A. System Modeling and Configuration

We use an execution-driven simulator McSimA+ [27] with detailed microarchitecture models as the backend for cores and cache hierarchy. For HMC memory modeling, we integrated a cycle-accurate simulator CasHMC [28] with McSimA+ to replace its memory system. We leveraged McSimA+'s Pin [29] based front end to implement *Active-Routing* instruction extensions. The microarchitectural behaviors of *Active-Routing* were implemented on the crossbar switch in HMC logic layer.

For power and latency modeling, we use CACTI [30] for on-chip cache power estimation, assume 5pJ/bit for each hop in memory network [31], 12 pJ/bit for HMC memory access [3].We implemented the *ARE* in verilog and synthesized it using TSMC 45 nm library. The multiplication takes the longest time, which is 6.61 ns, and the operand buffer takes 0.59 ns access time. As we use 1250 MHz for *ARE* and pipeline the arithmetic operations, it takes 9 cycles for each mult and 1 cycle for buffer access. At full load, *ARE*'s ALU can compute 1 FLOP/cycle. The area and power estimation is 0.02 mm^2 and 17.8 mW for ALU, 0.026 mm^2 and 16.9 mW for operand buffer, 0.05 mm^2 and 33.2 mW for Flow Table.

We configured the host CPU as a CMP with on-chip network and two level cache hierarchy with MESI coherence protocol. The 16 off-chip HMCs are connected to form a Dragonfly topology [6]. The system configuration evaluated in this work is shown in Figure 2 and described in Table I.

Table I
SYSTEM CONFIGURATIONS

Parameter		**Configuration**
CPU	Core	16 OoO cores @ 2GHz 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 $t_{CAS} = 9.9$ ns, $t_{WR} = 8$ ns, $t_{RP} = 7.7$ ns
	HMC	4GB/cube, 4 layers 32 vaults, 8 banks/vault
	HMC Network	16 cube DragonFly, 4 controllers Minimal routing, virtual cut-through 16 lanes link, 12.5 Gbps/lane CrossbarSwitch clock @ 1250 MHz
Active-Routing Engine	Flow Table	16 flow entries
	Operand Buffer	128 buffer entries
	Processing Element	1250 MHz clock frequency An arithmetic logic unit

B. 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 application 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 [32]. We summarize the workloads and applied optimization region as well as input data in Table II.

VI. EVALUATION

In this section, we evaluate ART-tid and ART-addr with respect to PEI [18], implemented by adding a computation unit at each vault controller supporting PEIs. It can compute a dot product of 2 4D 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 present performance evaluation followed by power and energy analysis. Then we show the potential of dynamic offloading through a case study.

A. Performance

1) Speedup: Figures 8 (a) and (b) show the execution time speedup of benchmarks and microbenchmarks, respectively. Both ART-tid/addr schemes create multiple trees from all 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 upto 7× speedup. In *sgemm*, almost all the execution time is spent in matrix multiplication. During the kernel execution, PEI needs to fetch one of the source matrices and also update the target matrix, causing read and write contention on the limited cache, which results in 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 significantly. 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 is better than ART-tid due to more balanced work distribution, which will be discussed in short. 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.

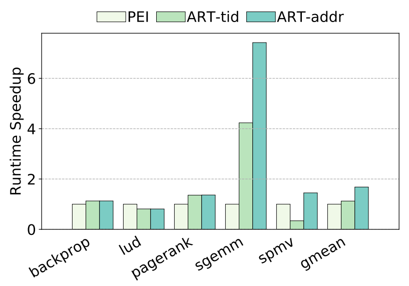
Figure 11 shows a heatmap of *spmv* for ART-tid and ART-addr. In the heatmap darker colors are used for denoting higher number of event occurrences. Each big square depicts the whole memory network and each small square block represents one cube in the memory network. In ART-addr, the work is evenly 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].

To evaluate scalability, we also run experiments for *mac* on 64-cube dragonfly memory network. 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 over PEI by 4.6× and 7.1×, respectively. When comparing each technique's performance on the two different memory networks for the same problem size, PEI

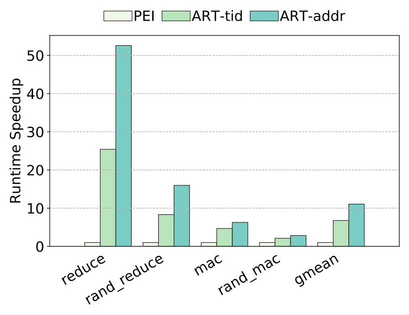
[1]The operand distribution are different due to the dynamic memory allocation.

Table II
WORKLOADS

Workloads		Optimization Region	Input Data Size
Benchmarks	*backprop* [33]	activation calculation in feedforward pass	2097152 hidden units
	lud [33]	upper and lower triangular matrix decomposition	4096 matrix dimension
	pagerank [2]	ranking score calculation	web-Google graph [34]
	sgemm [35]	matrix multiplication	4096x4096 matrix
	spmv [35]	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

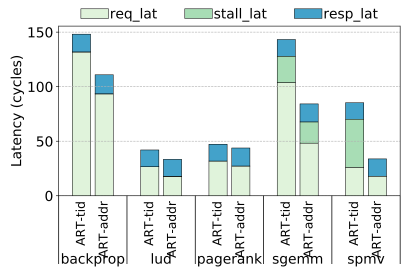


(a) Benchmarks

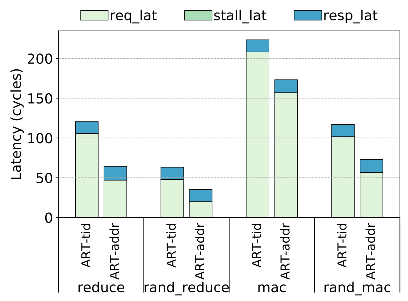


(b) Microbenchmarks

Figure 8. Runtime Speedup over PEI.

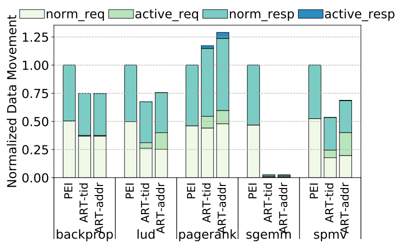


(a) Benchmarks

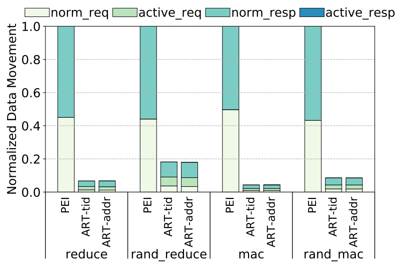


(b) Microbenchmarks

Figure 9. Update Roundtrip Latency Breakdown into Request, Stall and Response Latency.



(a) Benchmarks



(b) Microbenchmarks

Figure 10. On/Off-chip Data Movement Normalized to PEI.

incurs 2% performance degradation on 64-cube network compared to its performance on 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. On the contrary, ART benefits from both memory parallelism and network concurrency, therefore it tends to scale better for larger memory networks.

2) Update Offloading Round-trip Latency: In Figure 9, round-trip latency is broken into request, stall and response to understand the contribution of different communication components for **Update** offloading. As expected, the total latency is inversely proportional to the performance shown in Figure 8. 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 directory, thereby incurring less request latency. The stalls are mostly due to queuing in HMC controllers.

3) Data Movement: We evaluate data movement as the data size transferred between the host processor and memory network. The data movement breakdowns for normal data and active data transfer are shown in Figure 10. For most applications, ART-tid/addr can reduce the memory requests

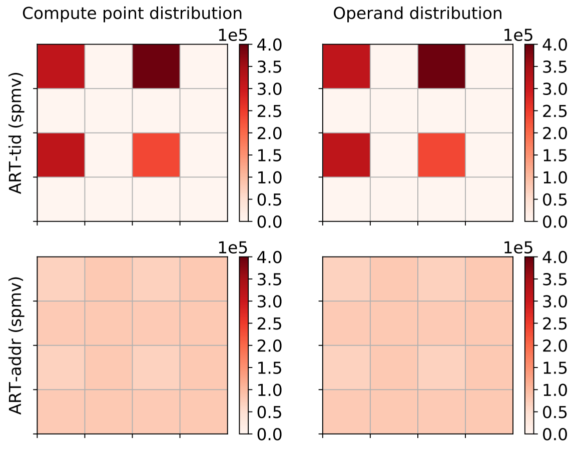


Figure 11. SPMV Compute Point and Operand Distribution.

fetching the data, mostly source operands, 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 the benchmarks, only parts of the whole parallel phase that we evaluate are our optimization targets. The other phases still require data movement. Another overhead comes from massive fine-grained offloading in this benchmark due to the irregular memory access pattern. Further preprocessing on the data [15] can solve this problem to gain performance and reduce data movement further.

In the microbenchmarks, the whole parallel phase can be optimized and hence the data movement decreases significantly. In *reduce*, the majority of its execution time 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 exhibit very good spatial locality in their memory accesses, which is exploited in cache-block grained offloading for vector processing. However in PEI, it 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.

B. Power and Energy

1) Power Consumption: We present the power consumption breakdowns into cache, memory and memory network in Figure 12. We observe 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. As a result, 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 CPU, thus PEI consumes more power. One exception is *pagerank*, which has irregular memory access patterns. ART offload computation *flows* in single operand granularity, causing high overhead in offloaded packets and operand packets, thus consuming more network power.

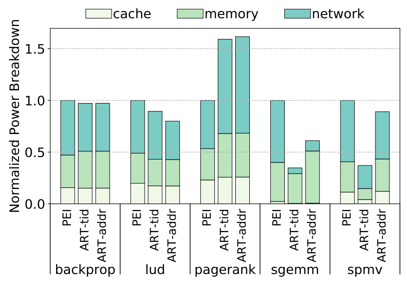
Microbenchmark *mac* has similar power characteristics as the benchmarks behaving regular memory access patterns. For *reduce*, ART-tid/addr can massively process the reduction near-data in memory cubes without moving data around, which leads to more intensive memory accesses and more offloading. For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

2) Energy Consumption: Figure 13 shows the energy consumption for cache, memory and memory network. 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 that has high overhead due to fine-grained offloading. For *sgemm* and microbenchmarks, energy consumption is reduced dramatically due to significant running time speedup. We gain enormous benefit as most parts of these applications can be optimized by *Active-Routing*.

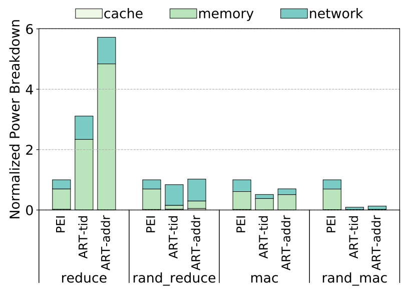
3) Energy-Delay Product: Figure 14 shows the normalized energy-delay product (EDP) over PEI in logarithmic scale to show the energy efficiency. We observe that ART-tid/addr has lower EDP for all applications except for *spmv* with ART-tid. The reductions in execution time as well as energy consumption contribute jointly 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 on *spmv* has lower EDP. To summarize, ART-tid and ART-addr reduce the EDP by 80% on average compared to PEI.

C. Dynamic Offloading: A Case Study

In this section, with the help of an example we show that the performance can be further improved using a runtime knob. The runtime knob dynamically decides whether to offload computations (**Updates**) on the basis of memory access and communications patterns to achieve more performance gains. Execution phases that exhibit good locality of data accesses experience performance benefits by exploiting cache hits when scheduled on the host processor. In *lud*, it decomposes a matrix into upper and lower triangular

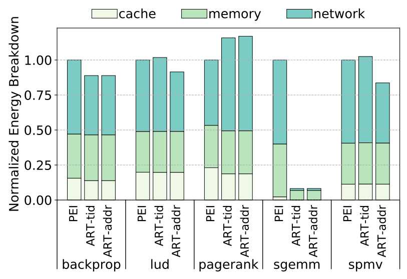


(a) Benchmarks

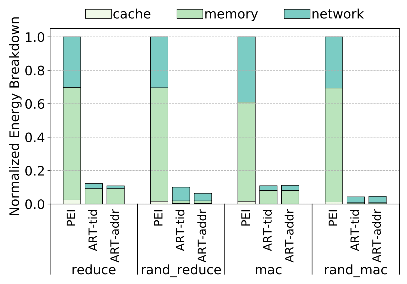


(b) Microbenchmarks

Figure 12. Normalized Power Consumption over PEI.

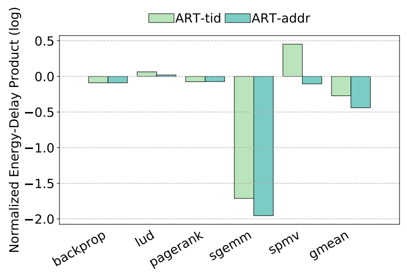


(a) Benchmarks

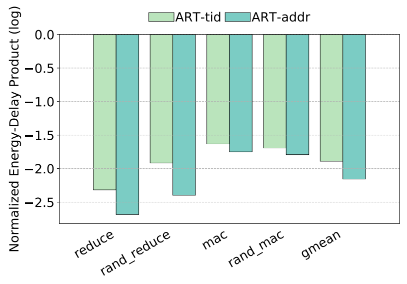


(b) Microbenchmarks

Figure 13. Normalized Energy Consumption over PEI.



(a) Benchmarks



(b) Microbenchmarks

Figure 14. Logarithmic Scale of Normalized Energy-Delay Product over PEI.

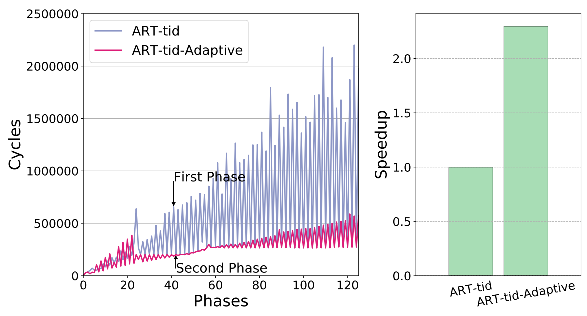


Figure 15. LUD Phase Analysis and Dynamic Offloading

matrices. Computations for these two matrices can be broken into two different phases. The first phase is for computations of the upper triangular matrix and the other is for those of the lower triangular matrix and they are executed iteratively. These two phases have different locality of data accesses. The second phase has a good data locality since its data access pattern is row-major order, whereas the data access pattern of the first phase is in column-major order.

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 Figure 15. 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.

VII. Additional Related Work

Near-Data Processing. Recently, NDP architectures is becoming an active research area in architecture community [18], [9], [19], [36], [11], [10], [12], [37]. Ahn et al. proposed Tesseract [9], a programmable PIM accelerator for large-scale graph processing. Nair et al. [36], [10] proposed Active Memory Cube (AMC) by leveraging HMC to place vector processing units in the logic layer. AMC suffers from delays due to instruction pre-loading as well as delay and energy overhead of its complex interconnection network. Most recently Fujiki et al. [38] propose a programmable in-memory processor architecture, and data-parallel programming framework using non-volatile memory. Mondrian [15] takes an algorithm-hardware co-design approach to sequence irregular accesses for better locality. Recent study [39] analyzed Google workloads and discovered the data movement as the bottleneck for performance and energy efficiency, which is also the problem *Active-Routing* tries to solve.

Processing in the Interconnection Network. Previous research [20], [21], [22] has encouraged interconnection networks to offer more functionalities other than just routing packets. Active Message [20] embeds the function pointer and arguments across the network to perform tasks in remote compute nodes. Pfister et al. [21] and Ma [40] proposed mechanisms to combine messages so as to reduce network traffic. Recently, IncBricks [22] implements an in-network caching middlebox for key-value acceleration in router switches. Several studies [23], [24], [25] proposed mechanisms to optimize shared value update or reduction in the network. The NYU Ultracomputer [23] implemented adder in network switches to coalesce the atomic fetch-and-update for the same target address along their way to memory. Panda [24] and Chen et al. [25] describe similar mechanisms that provide network interface functionality as well as hardware support for MPI collective reduction communication in a static manner. All of these mechanisms present varied solutions to support data processing in the network but still suffer the burden of data movement from memory to CPU, while *Active-Routing* solves this issue.

VIII. Conclusion

We propose *Active-Routing*, an in-network compute architecture, to accelerate reduction on data processing operations in data-intensive applications for near-data processing. *Active-Routing* is implemented as a novel three-phase processing schedule, 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 offload the computations in various granularities by exploiting their locality characteristics to reduce offloading overhead. Compared to the state-of-the-art PIM architecture, *Active-Routing* can achieve up to 7$\times$ speedup with a geometric mean of 60% performance improvement and reduce energy-delay product by 80% on average, showing promising potential for in-network computing and data-flow processing.

Acknowledgment

The authors would like to thank the shepherd and anonymous reviewers for their helpful comments and suggestions.

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)*, pp. 1097–1105, Curran Associates Inc., 2012.

[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)*, pp. 44–55, IEEE Computer Society, 2015.

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

[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*, pp. 432–433, IEEE, 2014.

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

[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)*, pp. 145–156, IEEE Press, 2013.

[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)*, pp. 1–14, IEEE, 2016.

[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*, pp. 481–492, 2017.

[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)*, pp. 105–117, IEEE, 2015.

[10] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien, and R. Nair, "Data Access Optimization in a Processing-in-Memory System," in *International Conference on Computing Frontiers (CF)*, p. 6, ACM, 2015.

[11] 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)*, pp. 113–124, IEEE, 2016.

[12] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 457–468, 2017.

[13] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-Stacked Memory + Logic Devices on MapReduce Workloads," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 190–200, 2014.

[14] A. F. Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 283–295, 2015.

[15] 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*, ISCA '17, pp. 639–651, 2017.

[16] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems," in *International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[17] K. Hsieh, E. Ebrahimi, 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," in *International Symposium on Computer Architecture (ISCA)*, pp. 204–216, IEEE Press, 2016.

[18] 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)*, pp. 336–348, IEEE, 2015.

[19] 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.

[20] 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)*, pp. 256–266, IEEE, 1992.

[21] 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.

[22] 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)*, pp. 795–809, ACM, 2017.

[23] 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.

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

[25] 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)*, pp. 1–10, IEEE, 2011.

[26] 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*, pp. 461–475, ACM, 2018.

[27] 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)*, pp. 74–85, April 2013.

[28] D. I. Jeon and K. S. Chung, "CasHMC: A Cycle-Accurate Simulator for Hybrid Memory Cube," *IEEE Computer Architecture Letters*, vol. 16, pp. 10–13, Jan 2017.

[29] 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)*, pp. 190–200, ACM, 2005.

[30] 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)*, pp. 3–14, 2007.

[31] 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)*, pp. 678–690, ACM, 2017.

[32] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. Patwary, M. Ali, Y. Yang, and Y. Zhou, "Deep learning scaling is predictable, empirically," *arXiv preprint arXiv:1712.00409*, 2017.

[33] 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)*, pp. 1–11, IEEE Computer Society, 2010.

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

[35] 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," tech. rep., University of Illinois at Urbana-Champaign, March 2012.

[36] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura, "Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, p. 17, 2015.

[37] 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)*, pp. 273–287, ACM, 2017.

[38] 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*, pp. 1–14, ACM, 2018.

[39] 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*, pp. 316–331, ACM, 2018.

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