ACTIVE-ROUTING: PARALLELIZATION AND SCHEDULING OF 3D-MEMORY VAULT COMPUTATIONS

An Undergraduate Research Scholars Thesis

by

TROY FULTON

Submitted to the Undergraduate Research Scholars Program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Eun Jung Kim

May 2020

Major: Computer Science

TABLE OF CONTENTS

SECTION Page

ABSTRACT

Active-Routing: Parallelization and Scheduling of 3D-Memory Vault Computations

Troy Fulton
Department of Computer Science & Engineering
Texas A&M University

Research Advisor: Dr. Eun Jung Kim
Department of Computer Science & Engineering
Texas A&M University

In an age where big data is more available than ever, new high-bandwidth, low-latency memory technology, such as Hybrid Memory Cubes (HMC), have extended into the third dimension to tighten the increasing gap between memory and CPU speeds. Processing power built into these new 3D memory technologies allows CPU cores to offload computations to memory, leading to recent interest in the design space of Processing-In-Memory (PIM) when several HMC units are chained together in a network. Using topology-oblivious Active-Routing technique in such a network, computations like dot products over a large set of data can be distributed across a virtual "tree" such that partial results are compounded at every branch "on the way" back to the CPU.

We propose driving performance of Active-Routing by offloading computations to memory with high throughput offloading techniques. We present Vault-Level Parallelism to further parallelize computations by strategically dispatching computations to DRAM vault controllers within each HMC. Our new implementation distributes the resources of Active-Routing to each of the vault controllers in the HMC so as to reduce contention for compute resources. We simulate our implemented techniques and assess their performance using previously developed micro-benchmarks and a widely accepted benchmark in scientific computing. The evaluation results show an increase in overall data throughout the Active-Routing Tree with an aggregate $23\times$ speedup.

ACKNOWLEDGMENTS

I would like to thank my research advisor Dr. E. J. Kim for her patience and careful attention to my work, as well as her supportive guidance throughout my first research experience. I would also like to thank my mentor Jiayi Huang, who has given me so much guidance and support as I learned how to use the tools necessary for this research. I also thank Sungkeun Kim, Pritam Majumdar, and all of the current or former researchers in the High Performance Computing Lab who have contributed to this work and provided me assistance.

My thanks also goes to my friends and other colleagues, as well as the department faculty and staff, who have contributed to making my time at Texas A&M University excellent and enjoyable.

SECTION I

INTRODUCTION

For much of the history of computer architecture since the mid-1980s, there has been a significant gap between CPU latency and memory latency [1], [2], [3]. Trends in CPU core speeds have shown an exponential increase, while memory speeds show a much slower growth, leading to a widening gap over recent decades [1]. Despite this deficit, recent improvements in technology and the emergence of several networked devices put data availability at an all-time high, and the need to process huge sets of data can be seen in several modern applications. Kernels in graph processing [4], [5] and neural networks [6] for industries such as social networks [7], cognitive computing [8], and warehouse-scale computing [9] have demonstrated the need for simple computations over large data sets with low data reuse [10], [11]. These kernels tend to generate excessive data movement throughout cache and main memory, which wastes precious energy and bandwidth and puts pressure on the communication fabrics in the memory hierarchy. Due to comparatively high processor speeds and these trends in program behavior, modern computer system designs now focus on maximizing memory retrieval efficiency and minimizing data movement across the system.

Efforts to increase bandwidth and decrease latency of memory accesses have advanced memory technology since the latest generations of DDR DRAM [3], [12], [13]. New 3D memory stacking technologies, such as Hybrid Memory Cubes (HMC) [2] and High Bandwidth Memory (HBM) [14], have been introduced to increase memory bandwidth and capacity in a cost-effective way. An HMC, for example, avoids the bandwidth bottleneck of the DDR DRAM bus by making use of DRAM stacking and Through-Silicon Via (TSV) technology to provide vertical slices of logically independent "vaults" that can operate concurrently [2], [3], [13]. Die-stacked memory also provides an abstraction of memory that scales well for dense, high-capacity memory. In the traditional DDR DRAM architecture, the memory controller accesses DRAM chips through busses

that must be reconfigured each time a new DRAM chip is added. However, since HMCs contain built-in logic hardware at the base of their DRAM stacks, their controller logic can be expanded to route data to and from other HMCs. Thus, HMCs can be chained together to form a packet-switched network capable of delivering scalable-capacity and high-bandwidth memory.

To scale such bandwidth provisions to larger multi-core systems with many HMCs, the traditional processor-centric design does not suffice. Kim, et. al. [12] suggest a memory-centric interconnect design where all communication between cores routes through a shared network of HMCs and a hybrid architecture where processors make use of both their own communication fabric as well as the pool of shared memory. Although the memory-centric design often makes the best use of both memory and processor bandwidth when compared with the processor-centric and hybrid approaches, it still suffers from long multi-hop latency between cores and puts pressure on the memory interconnection fabric because of excessive data movement [15].

Moreover, energy consumption due to data movement hinders processors from achieving full computational potential. To solve this problem, recent research [5] has suggested moving processing power closer to where data is stored. Throughout the memory hierarchy, Near-data Processing (NDP) has been proposed as a paradigm for installing minimal computation hardware close to data storage locations, such as main memory [5], cache [10], or persistent storage [16]. By providing processing power with direct access to DRAM, HMCs have realized one instance of NDP known as Processing-In-Memory (PIM) [4], [9], [11], [5], [17], [18], [19], [20], which suggests that processing hardware should be integrated on the same module as memory [2]. Offloading computations to memory has become a growing research area, and the proposed methods for offloading computation to memory widely vary by application. Proposed techniques for implementing PIM with existing 3D-Memory technologies include designing additional NDP accelerator hardware in each HMC [5], [17], [19], [11], offloading instructions from the CPU to memory [4], [18], and extending the Instruction Set Architecture (ISA) to PIM instructions [21], [20], [22]. The computation takes place in memory while the CPU is free to attend to other tasks until the result comes back, thereby mitigating the communications across the long, multi-hop path between

memory and the CPU. These techniques perform well on irregular memory accesses for simple, atomic, one- or two-operand operations because of their reduced communication traffic and energy footprints [4], [11], [5], [18]. However, they do not perform well when data are scattered across the network in different modules, since sequences of fetch operations for data in different modules incur communication overhead and data movement in the interconnection network.

Several previous studies [11], [23] have proposed enhancing routing capabilities in the communication network to improve the efficiency of a wider variety of memory access patterns. As early as 1982, the NYU Ultracomputer [23] introduced a way to coalesce fetch-and-add synchronization operations in a multi-node network by implementing adders in routers. In 1992, Eicken, et. al. [24] proposed Active Messages as a way to encode the address of a handler function that can quickly retrieve data in a message to a remote compute node, rather than inflating every packet with a data payload. More recent proposals for communication fabric expansions include efforts to optimize MPI collective communications [25], [26]. These proposals work well for a few atomic operations, such as the MPI collective *allreduce*, but the modern demands of machine learning require more complex and diverse operations, such as a dot product [7]. Kwon, et. al. recently proposed MAERI [27], a configurable Deep Neural Network (DNN) accelerator that maps data-flows in DNNs to reduction trees across the network to improve efficiency by reduced data movement. The solution is limited, however, by the tree structure of the topology, as the multiplications in the dot product can only be done in the leaves of the trees of the topology.

Huang, et. al. introduced "Active-Routing" in 2019 as a topology-oblivious routing architecture for generalizing and accelerating reductions of data-flows in a dynamically constructed routing tree [28]. In Active-Routing, a compute kernel is mapped to a reduction across intermediate operators. That kernel is then dynamically constructed as a virtual tree in the network and computed as data flows up the tree to return to the root. The routing algorithm involves dispatching packets from the CPU to the network in three phases: (1) tree construction, where the compute nodes used for reductions are recorded, (2) update, where in-memory computations take place, and (3) gather, when reductions are sent up the tree towards the root of the reduction tree. To

implement such a method, Huang, et. al. [28] show that routing logic in the intra-cube network of each compute node can be expanded with an "Active-Routing Engine" (ARE), whereby "Active-Routing Flows" created for each Active-Routing tree store the parent to which to send the partial result. The Active-Routing Engine also stores a set of operand buffers needed for the computation as data are retrieved from memory and queued for computation. In order to compute a result over a set of data in one memory cube, the steps involve registering a flow for the computation, requesting each operand from the corresponding vault(s) one at a time as request packets come in from the processor, filling the operand buffers with the response, computing the partial result, and sending the result to the parent of the tree when requested. Active-Routing shows that runtime can be improved by up to $7\times$ over state-of-the-art PIM technologies [28] by performing computations to aggregate data in the network "on the way" back to the CPU.

The Project

Despite the highly parallel nature of Active-Routing trees, high overhead incurred from offloading limits Active-Routing from taking full advantage of the compute potential of the silicon area available in the logic layer of each HMC. Instruction offloading in the current Active-Routing implementation limits throughput due to the high overhead of fine-granular offloading, making Active-Routing requests sparse in the memory network. Furthermore, its computing throughput is limited by the single compute unit in each cube, leaving abundant spare areas vacant. To this end, we propose kernel and page-level offloading as well as Vault-Level Parallelism to solve these problems.

To reduce the instruction offloading overhead and improve the offloading throughput, we first propose offloading kernels with page granularity to the memory network. In this project, we implement a method for offloading the complete Active-Routing compute kernels to the Memory Controller to avoid congestion from many processor cores to a few memory controllers traffic pattern. To further increase the volume of computations embedded in a single packet, we introduce page-granular offloading to embed several cache lines in a single packet.

Secondly, we propose a new Vault-Level Parallelism architecture to improve the compute

throughput by making use of the additional area available in the logic layer. In the new architecture, the ARE serves as a coordinator for the vault compute units in each cube. Vault controllers and the ARE form a one-level sub-tree for intra-cube reductions, which distributes the compute burden on each cube. Due to the increased compute power and higher dispatching rates offered by the VLP architecture, VLP yields a significantly higher throughput than baseline.

Our evaluations show that kernel offloading leads to up to $4.5\times$ runtime speedup over [28]. The highest offloading throughput comes from page-granular offloading, which gives $7\times$ and $9\times$ improvement in runtime over kernel offloading. Moreover, Vault-Level Parallelism with page-granular offloading yields up to a significant speedup of $23\times$ over the architecture in [28].

SECTION II

BACKGROUND

In this section, we explain the properties of Hybrid Memory Cubes and their roles with Interconnection Networks. Then, we describe the properties of Processing-In-Memory necessary to allow us to implement our Vault-Level Parallelism technique.

3D Memory Stacking

Although the concepts of DRAM stacking and Through Silicon Via (TSV) technologies date back to the 1980s and 1960s, respectively [13], Micron's recent realization of die-stacked memory technology [2] has given way to a denser, more energy-efficient memory solution. Several recent studies take advantage of this trait to study the effects of PIM in HMCs [5], [11]. The two main competing 3D Memory techniques- Hybrid Memory Cubes (HMC) [2] and High Bandwidth Memory (HBM) [14]- use TSVs to achieve high-bandwidth, low-latency communications between a logic layer and layers of DRAM stacked on top. Without loss of generality, our implementation makes use of Hybrid Memory Cubes as the underlying technology, but these techniques can also be demonstrated on other die-stacked memory technologies, such as HBM, and other interconnects.

Figure 1 shows the configuration of an HMC. As introduced in 2011 [13], HMCs are partitioned vertically into vaults, each corresponding to a vault controller in the logic layer and several TSV connections [2]. Each HMC logic layer contains its own intra-cube network connected to a high-speed network fabric local to the cube, connecting a number of I/O interfaces and the vault controllers internal to the memory cube. In [28], the ARE is placed in the intra-cube network so that it can control the flow of Active-Routing packets in its own cube and record the flow of packets to and from other HMCs.

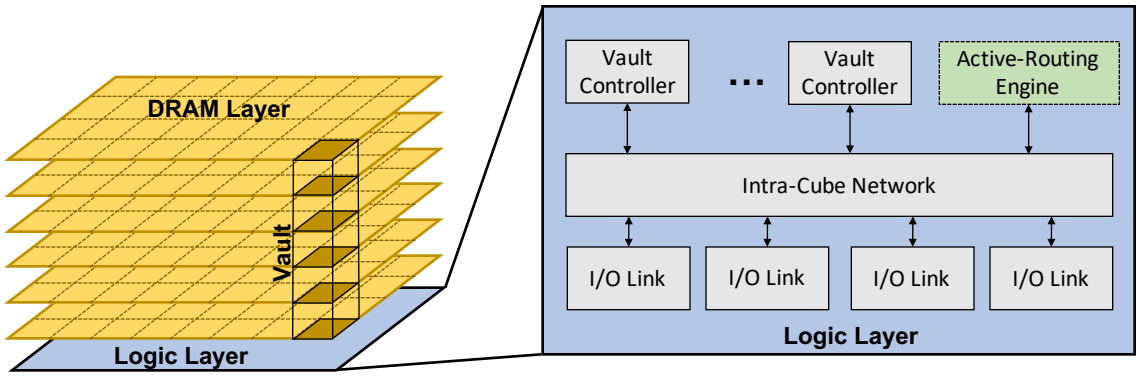


Figure 1: Hybrid Memory Cube Configuration

Vault Controllers are spread out across the logic layer and leave plenty of vacant silicon area for additional logical hardware. Some research studies have shown uses for minimal hardware logic at the vault or cube levels [21], [4]. For example, [18] places a small amount of hardware logic for each Linked-list engine in each vault to exploit locality in the Linked-list nodes. On the other hand, some research has shown efficiency improvements in using accelerators with much more processing power [5], such as the Mondrian Data Engine, which places an ARM Cortex A35 core, 1024-bit SIMD unit, stream buffer, and object buffer in every vault [11].

It should be noted that packets can be sent in any direction through the Intra-Cube Network. The CPU exchanges request and response packets with HMCs to access and store memory, and in the Intra-Cube Network, any such packet can enter from any port or vault and be routed to any other port or vault. This feature enables the HMC to exercise routing functionality in the logic layer so that several HMCs can be chained together in a network. In [28], the ARE is placed in the Intra-Cube Network to monitor the flow of packets to and from its cube and to control the flow of data in a particular cube.

Interconnection Networks

On-Chip Networks

In recent years, there has been an increase in demand to move to multi-core architectures in a variety of fields, including IoT, mobile devices, and high-end servers [29]. By nature, these

multi-processor systems demand high-bandwidth, low-latency communication between cores. Traditional systems with 8 or fewer cores perform well using a bus or a crossbar switch. However, for systems with increasing core counts, neither the bus nor crossbar will meet communication requirements because of their poor scalability in terms of area, power, and performance [29]. To meet these communication requirements, an On-Chip Network (or Network-on-Chip) that makes use of interconnected switches has been adopted as a scalable solution.

In an On-Chip Interconnection Network, each communicating entity with an interface to the network is a node, and the organization of connections between nodes is known as the topology. Topologies are often automatically synthesized to optimize communication for specific applications [29], but there are also several well-known, generic topologies, such as ring, mesh, and taurus. For example, **Figure 3** shows a mesh topology, where each node is connected to its 2, 3, or 4 neighbor nodes.

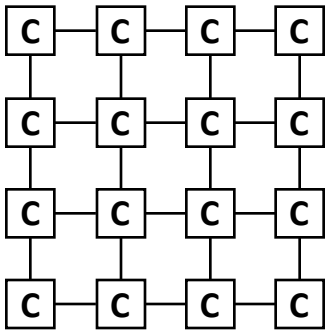


Figure 2: Mesh Topology (C blocks are Compute Nodes)

When one node needs to communicate with another node in the network, it forms a message, encodes the address(es) of the receiver(s) and perhaps other metadata in the message, and inserts the message into the network. The network then decodes parts of the message to find the recipient and delivers the message to the target node. In the mesh configuration in **Figure 3**, a com-

pute node C can only communicate with its immediate neighbor, so if two nodes are not neighbors and would like to communicate, they must communicate only through other intermediate node(s) in the network.

When designing this interconnection fabric, designers often need to make choices related to the expected performance. For example, the previously mentioned crossbar interconnect for bidirectional communication between n entities is a $n \times n$ matrix of switches, which provides sufficient bandwidth for a small number of communicating nodes, but when scaling networks to a higher number of inputs, there is a strong motivation to use fewer connections. [29] shows that topologies such as butterfly, ring, and torus cut down on power and provide better network throughput for high volumes of traffic.

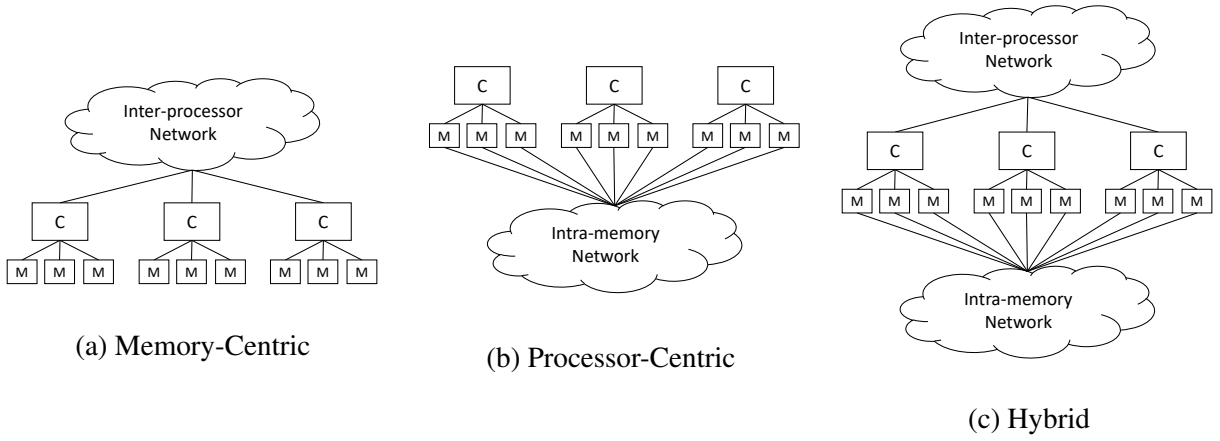
In reality, nodes often divide messages into individually addressed packets with meta-data, which are routed through the network independently. This type of network fabric is known as a packet-switched network. Routers in the network receive packets, read the meta-data to determine where to send the packet, and transmit the packet either to the next router on the packet's route to its destination or to the packet's destination node. When a router receives a packet, its routing algorithm employed in the architecture tells the router how to decide which router to send a particular packet next. Much of recent research into how to improve the performance of On-Chip Networks involves improvements to this algorithm so that more can be computed in memory [28], [26], [27].

Similarity with Off-Chip Networks

It should be noted that although On-Chip Networks borrow ideas from Off-Chip Networks, On-Chip Networks are subject to different constraints [29]. In On-Chip Networks, the abundance of wires alleviates the bandwidth bottlenecks of Off-Chip Networks, and the long I/O delays of transmitting messages off-chip are mitigated. However, in On-Chip Networks, the power and area constraints are much tighter, since the network competes with the cores for silicon area. Unlike the routers in the Internet, routers in an on-chip network do not often have the luxury of hardware support for much complex functionality, such as monitoring traffic across the network.

Memory Networks

In a similar manner, HMCs can also be organized into interconnection networks so that HMCs can route requests and responses between the CPU and memory to and from other HMCs. Since modern memory requirements would require more DDR DRAM capacity than most processors have pins to support, it becomes cost-effective to chain HMCs together in their own memory network. Since HMCs are packaged with space for routing functionality support, there is a distinction to be made between memory-centric network designs and processor-centric designs [12]. In traditional processor-centric designs like the one shown in **Figure 3b**, memory connects only to its associated processor, so when a processor needs memory from another processor, it must first request the memory from the connected processor.



(a) Memory-Centric

(b) Processor-Centric

(c) Hybrid

Figure 3: Network Designs in [12] for (M) memory- or (C) processor-centric networks.

Memory-centric designs like the one in **Figure 3a**, on the other hand, connect a chain of HMCs to form the entire network. All inter-processor and inter-memory communications route through the shared "pool" of memory. A recent study [12] has shown that memory-centric designs for networks of HMCs lead to better overall throughput provisioning than the typical processor-centric designs. Our system design adapts the hybrid architecture described in [12] shown in **Figure 3c** in which processors form their own Network-on-Chip for inter-processor communication separate from the memory network connecting HMCs.

As in [28] and [2], the HMC uses its four I/O Links to communicate with other compute nodes. The memory network is controlled by a number of memory controllers, which connect to the links of one or more HMCs in the memory network as gateways for traffic between the memory controller and the Network-On-Chip. Routing logic implemented in the logic layer gives each HMC the processing capabilities to forward packets to the correct HMC and vault in the network. Thus, when an HMC receives a request for data from one of its ports, it can route the packet either to one of its vaults (when the data are present locally) or to the link connected to the next HMC on the path to the destination HMC. When the response holding the requested data is ready from the vault, it is sent back to the intra-cube network, and the routing logic sends it to the link that leads back to the memory controller, which will deliver the data to the CPU.

SECTION III

RELATED WORK

In this section, we describe Active-Routing [28] to form a basis for our techniques. We then discuss other recent, related techniques for PIM and processing in the Interconnection Network.

Active-Routing

PIM-Enabled Instructions

Ahn, et. al. introduced PIM-Enabled Instructions (PEI) [21] as a set of instructions that can be added to the architecture (accompanied by hardware additions to the memory technology). The architecture for PEI maintains both the illusion that the program executes sequentially and that the processor executes all instructions, although the architecture dynamically decides when to process in memory based on the locality of the data. Their evaluations show an overall increase in system performance, and building upon PEI, Huang, et. al. [28] propose Active-Routing for further optimizations in memory network.

The Architecture

Figure 4 shows the system used in [28] to demonstrate Active-Routing. In this configuration, HMCs form a dragonfly network, and four HMCs connect to the Host CPU (left) via the HMC Memory Controller. The out-of-order (O3) cores on the host CPU each have a Network Interface (NI) to the NoC, through which they can send packets to the HMC Controller. The HMC Controller can then convert and route the packet to its first HMC in the network.

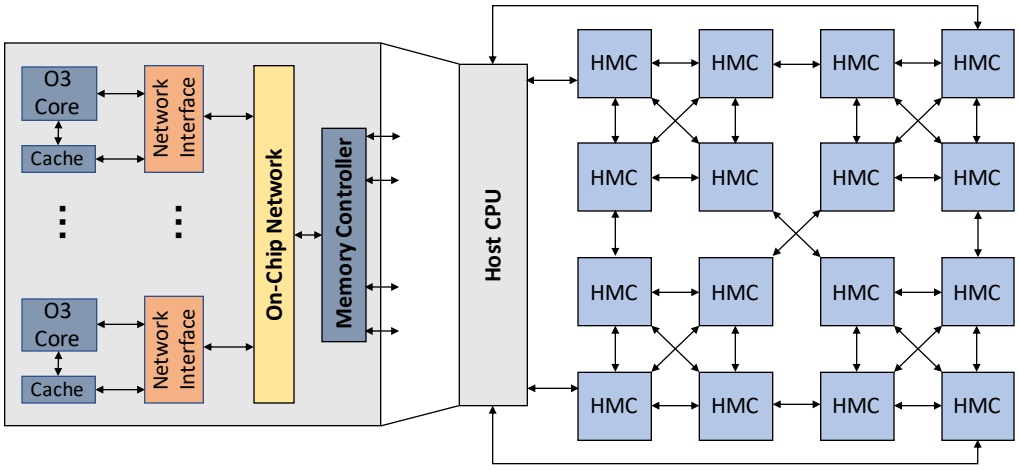


Figure 4: System Configuration for Active-Routing

In general, an Active-Routing flow (henceforth, a flow) maps a compute kernel to a reduction over the network. For each flow, an Active-Routing Tree (ARTree) is logically constructed in the memory network such that each node of the tree serves as a compute point for a partial result of the reduction. Each compute point is responsible for reducing the computations of its sub-tree and reporting the result to its parent. Flows are each given a unique flow ID, and when the reduction is done, the final result is routed back to the CPU. **Figure 5** shows an example of an ARTree for one dot product kernel (flow) in the same configuration as **Figure 4**.

Using the pseudocode in **Figure 5** as an example, we now explain the architecture and packet processing algorithm of Active-Routing. The host CPU dispatches two types of packets to the memory network. An `Update` packet contains the address(es) of the operand(s) needed for computation of a partial result, as well as the operation to be done on the operands and the Flow ID. In **Figure 5**, each $A_i \times B_i$ pair is offloaded to the memory network as an `Update` packet. Once the `Update` packet stream ends, a `Gather` packet is sent to the network as a signal to reduce all partial results in the network on the way back to the CPU when the partial results become ready. The algorithm handles these packets in three phases: (1) ARTree Construction, (2) Update, and (3) Gather.

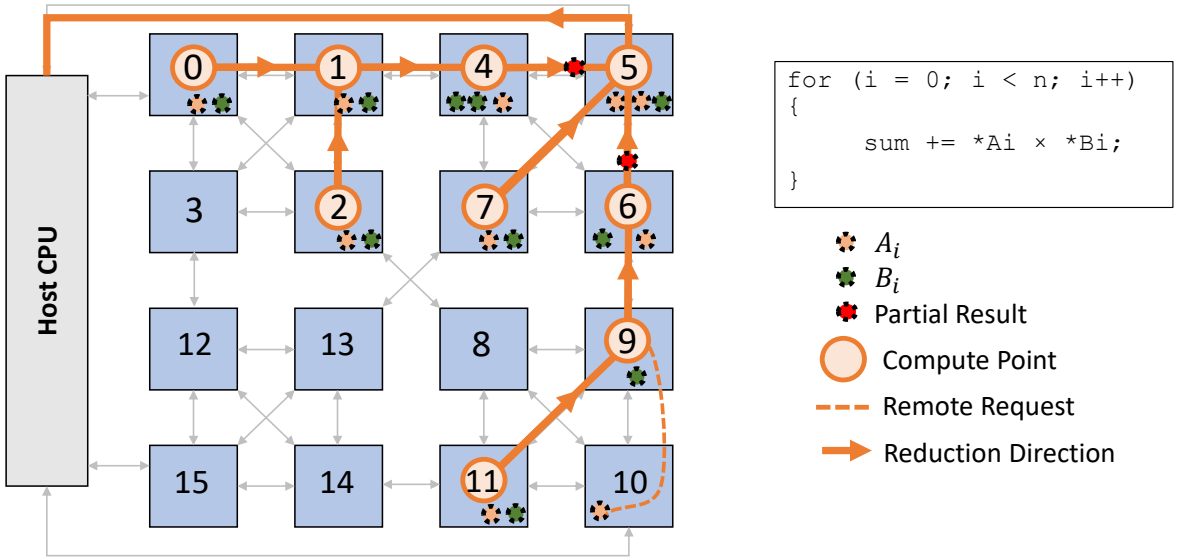
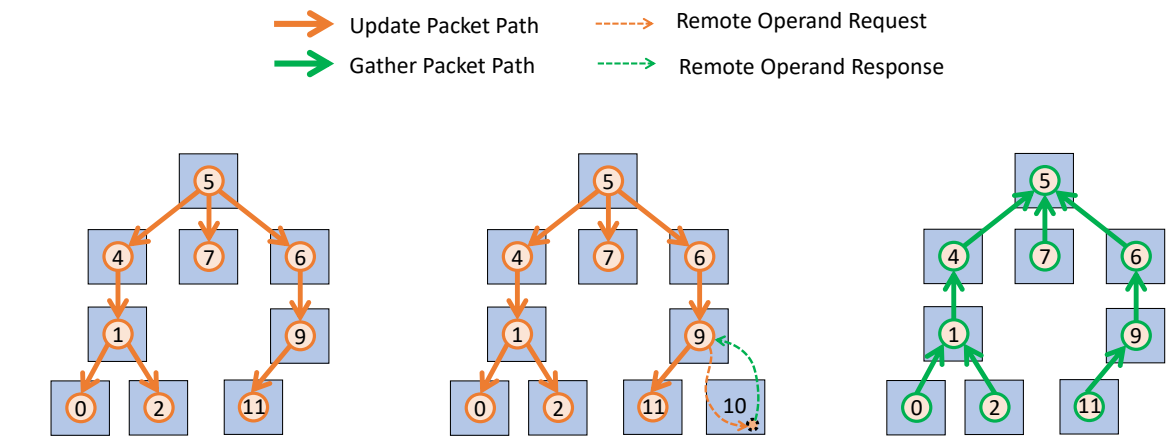


Figure 5: Example Dot Product Active-Routing Tree and Accompanying Pseudocode

Figure 6 shows the three phases of packet processing using the abstract ARTree from **Figure 5**. In the ARTree Construction Phase (**Figure 6a**), `Update` packets are sent to the memory and routed to the addressed cube. The ARTree is constructed along the paths that `Update` packets take as they are routed to their compute point in the network. For example, all `Update` packets in **Figure 5** enter through cube 5. Operands in cube 2 are routed through intermediate cubes 4 and 1, as shown in 6a. For cases such as that in **Figure 5**, where two operands reside in different cubes, the compute point is chosen as the node where the routes for the two operands diverge. When an `Update` packet is scheduled for a cube, that cube records the flow ID and its parent cube. If an `Update` packet reaches a cube it is not scheduled for, the packet is routed to its scheduled compute node, and the child compute node is recorded. Thus, the ARTree is recorded from each cube storing its own parent and children information.



(a) ARTree Construction Phase (b) Update Phase (c) Gather Phase

Figure 6: 3-Phase Packet Processing for Active-Routing Trees [28]

The Update Phase (**Figure 6b**) occurs in parallel with the ARTree construction phase as `Update` packets reach their destinations. In the Update Phase, operands are fetched from memory and their partial results are scheduled to be computed in memory. Each $A_i \times B_i$ product requires two operands A_i and B_i. When the two operands reside in the same memory cube, that cube fetches the operands locally. However, in cases like the operand that resides in cube 10 of **Figure 6b**, the operand(s) that do not reside in their compute point need to be fetched before the computation can be scheduled during the Update phase. Once the product $A_i \times B_i$ has been computed, it can be added to the partial sum in the compute point for that particular flow.

Finally, in the Gather Phase (**Figure 6c**), a `Gather` packet from the CPU serves as a signal that the stream of `Update` packets has ended and that the Update phase is over. When each node of the ARTree receives a `Gather` packet, it makes a copy of the packet to send to each of its children. When each node finishes the Update phase, it replies to its parent with its partial sum. When a node has received partial sums from all of its children, it sums the partial result, sends the partial sum of its subtree to its parent, and commits and de-registers the flow.

In [28], the authors note that access patterns when fetching operands can significantly affect performance. In the pseudocode for **Figure 5**, $*A[i]$ and $*B[i]$ dereference pointers independent

from any underlying data structure. Data structures such as vectors tend to follow regular access patterns, meaning that the data are spacially local to each other and can often be found in the same cache block(s). When access patterns for both operands are regular, the authors refer to the pattern as *regular-regular* and make use of the locality by offloading the computation at the granularity of a cache block for vector processing. However, when both sets of operands come from more complex structures, such as a graph [4] or linked list [18], they cannot often be found in the same cache block and are referred to as *irregular-irregular* patterns. These access patterns incur large overhead with `Update` packets for every operand and contribute significantly to network traffic power usage. In a *regular-irregular* access pattern, only one data set follows irregular accesses and can be loaded into spatially local storage before being processed with the regular data at the cache block granularity. For memory access patterns that follow regular-regular or regular-irregular access patterns, the overhead of packet metadata can be amortized by offloading multiple fetches in one packet.

Implementation

In this subsection, we first describe the ISA extensions and changes to the user-facing programming interface from Active-Routing. We then detail the hardware additions suggested to support Active-Routing, including the Network Interface and ARE. Finally, we describe the schemes used to enhance performance for Active-Routing.

Programming Interface and Extension to ISA

The programming interface implemented in [28] provides an API for the extensions made to the ISA for offloading `Update` and `Gather` packets to memory. The programming model communicates with the NI so that the NI can construct the packets and send them to the Memory Controller. **Figure 7** shows the Update and Gather APIs available to the user. The user supplies RR, RI, or II to specify which datasets are regular or irregular, the sources for the operation, the target address for the reduction, and the operation to be decoded by the HMC. Gather calls contain only the target address needed to decode the flow ID and the number of threads dedicated to the flow. This parameter is added as an implicit barrier, as the Memory Controller will not proceed to

send the `Gather` packet to the network until it receives a `Gather` packet from each thread for that flow. The API generalizes the op parameter to support simpler operations like move. The API is translated by the compiler to the extended instructions.

```
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);
```

Figure 7: Programming API for Offloading `Updates` and `Gathers` for Active-Routing [28]

Hardware Components

The three main hardware components that drive Active-Routing are the NI and Memory Controller in the On-Chip Network and the ARE in each HMC. Here, we describe the architectural hardware used to implement Active-Routing.

In the On-Chip Network, each core has access to the network through their NI, which handles the formation of packets and transmission to and from the network. In the NI, there are dedicated registers added for the purposes of Active-Routing that hold the opcode and operands for the NI to read when forming the packet. When these packets reach the HMC Controller, the HMC Controller makes the first routing decision of where to put the root of the ARTree and holds the final result if there are multiple roots.

The ARE located in the logic layer of each HMC both processes packets as they pass through the Intra-Cube Network and stores the state of the algorithm, including the ARTree information, flow state, and operand buffers, as seen in **Figure 1**. **Figure 8** details each component of the ARE. The Packet Processing Unit is responsible for decoding `Update/Gather` and operand response packets, generating operand request packets, and committing the partial result to the flow when ready. The Flow Table stores entries that record the state of the flow at any given time, including the partial reduction to be committed, counters for the number of operand requests

and responses to operand requests have been issued or received, respectively, and a gflag bit that records whether or not the `Gather` packet has been received. Each flow commits its partial result only after both the gflag has been set and there are no operand buffer entries left that are scheduled to compute for that flow. Operand Buffers serve as temporary storage for operands that have been fetched from memory and are scheduled to be processed as `Updates`. The ARE uses a pool of operand buffers to minimize their overhead and makes sure to reserve operand buffers before issuing the operand requests to memory so as to avoid deadlocking requests for operand buffers between flows. Operand buffers store their flow ID, opcode, operands, and a bit for each operand's status (whether it is ready or not) so that it can be scheduled for computation only after operands have been fetched. Finally, the Arithmetic Logic Unit (ALU) provides lightweight compute power with operations such as bitwise operations, add/multiply for integer and floating point operations.

Computation scheduling is done through queues holding the ID of the operand buffers in use. When all operands in an operand buffer entry become available, the ID of the operand buffer is enqueued in a ready queue. The ALU uses the ready queue to compute whatever operand buffer ID is at the head of the queue. Then, when the computation is done, the operand's ID is enqueued into a queue of free operand buffers, from which the ARE dequeues IDs when allocating new operand buffers on receipt of an `Update` packet.

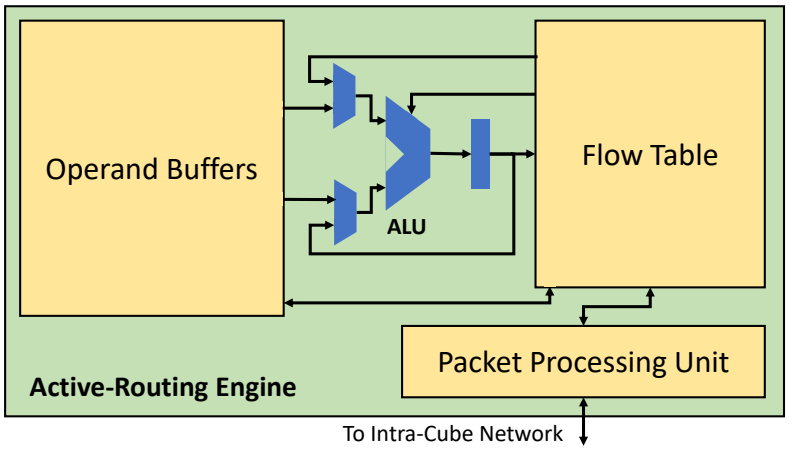


Figure 8: Active-Routing Engine

Figure 9 shows an abstracted view of the implementation for ARE as it is integrated with the rest of the HMC in [28]. The HMC in **Figure 9** can be seen as a detailed expansion of cube 5 from the example in **Figure 5**. For simplicity, each operand buffer entry is shown with only its operands, and the Flow Table entries are shown only with their partial result and a pointer to the parent cube in the ARTree.

When cube 5 receives an `Update` packet containing the addresses of A_1 and B_1 and a *multiply* opcode, it first decodes the packet in the Packet Processing Unit. If these are the first `Update` packets from this flow to reach cube 5, the ARE records the flow ID in the Flow Table, along with the parent node and the opcode for the flow. Since cube 5 is the root of the ARTree, it records the link to the HMC Controller as its parent. In the Flow Table, the response counter for `Update` packets is incremented. The ARE then reserves one operand buffer entry for the pair A_1 and B_1 and records the associated flow ID and opcode. Once the operand buffer entry has been reserved, the Packet Processing Unit dispatches the requests to the local vaults where A_1 and B_1 reside. If no operand buffer entries are available when the `Update` packet is processed, the ARE stalls until one becomes available.

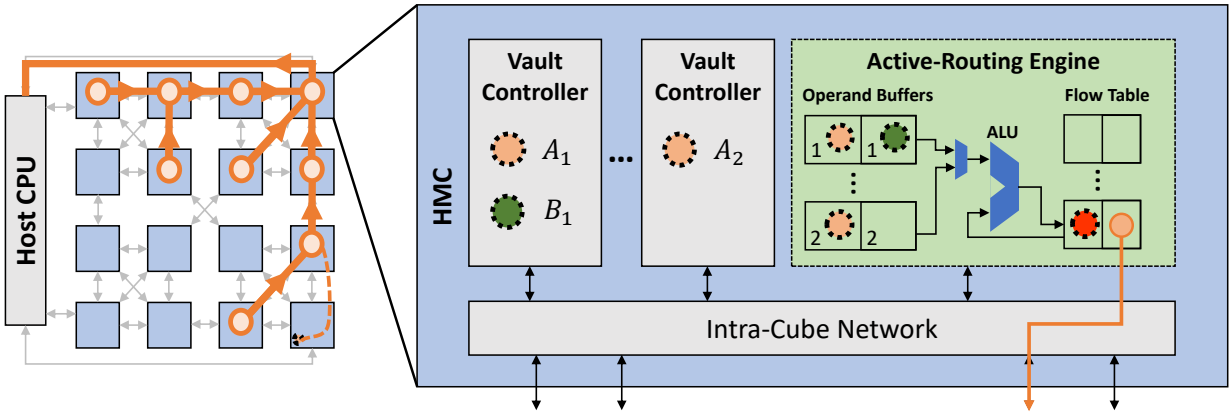


Figure 9: Abstracted view of Active-Routing Engine integration with HMC. The left Flow Table stores the partial result, and the right entry entry points to the parent node in the ARTree.

When operands A_1 and B_1 are fetched from the vault controller, their ready bits are set,

and the operand buffer entry is enqueued to the queue of operand buffers ready for computation. When the operand buffer reaches the head of the queue, the ALU is directed to compute $A_1 \times B_1$. Once the computation finishes, the result updates the partial result in the Flow Table entry, and the operand entry is freed. The Flow Table then increments the response counter.

For the purpose of discussion, we assume that, in the case of $A_2 \times B_2$, B_2 is located in cube 4. In this case, cube 5 will be chosen as the compute point upon receipt of the packet with $A_2 \times B_2$. Then, the Packet Processing Unit will follow the same procedure as it did for $A_1 \times B_1$, only now, the operand request for B_2 will be sent to cube 4. In cube 4, the Packet Processing Unit will decode the packet as a request for an operand and forward the packet to the appropriate vault. When the vault returns the operand to the Intra-Cube Network, it is routed back to cube 5, where it is processed in the same manner as with $A_1 \times B_1$. Unfortunately, this type of situation can lead to long stalls in the HMC compute point because the operand buffer entry must stay reserved for the request and response packets' journeys between cubes, which is much longer than journeys between vaults of the same cube.

Finally, when the `Gather` packet is received at cube 5, the Packet Processing Unit decodes the flow ID and replicates the packet to immediately send to its recorded children. Note that with the examples of just $A_1 \times B_1$ and $A_2 \times B_2$, there are no recorded children nodes yet. When `Update` packets pass through a given cube to another compute point, the Flow Table entry's request count is incremented, and when the given cube receives the `Gather` response, it increments its response count. The Packet Processing Unit then sets the gflag for that flow's Flow Table entry to indicate the initiation of Gather phase, and it waits until the request and response counts for the flow are equal to commit the flow and send the updated result to its parent. As the cube receives responses from its children, it updates its Flow Table partial result. Once the result has been sent to the parent, the Flow Table entry is de-registered.

Enhancement Schemes

Two important factors that affect the performance of Active-Routing are how to dynamically choose the root of the ARTree and offloading granularity. The authors of [28] present three

schemes for enhancing Active-Routing by taking into consideration these performance factors.

The first performance consideration is where to place the root of the ARTree. The four most obvious choices are those cubes who are linked with the Memory Controller (cubes 0, 5, 10, and 15). A first approach might be to statically assign one cube to be the root for any flow, such as cube 5 in **Figure 5**. This approach is referred to as Naïve-ART. In the enhanced ART-tid scheme, each of the four corner HMCs are considered for the root of the ARTree. Each worker thread that generates `Update` packet streams uses their ID to determine which cube should be the root of the ARTree so that flows are more evenly spread across the network. However, this approach is not always optimal because it tends to generate deep trees when data are not close to the chosen root. To take advantage of data locality, the ART-addr scheme simply sends all `Update` packets to the cube closest to its destination. ART-tid and ART-addr can create up to four trees for the same flow, leading to a better distribution of traffic in the network in most cases.

Another performance consideration is offloading granularity. To reduce the overhead of packets requesting data fetches and the overall number of data accesses, packets can be offloaded with cache-line granularity. Spatially local data tend to be located in the same cache block in memory, so it wastes fewer cycles to fetch a cache block than it does to fetch each operand serially. Benchmark testing in [28] shows that when combining the techniques of cache-line granular offloading with ART-tid and ART-addr, the Naïve-ART approach leads to worse performance than the baseline approach without any Active-Routing, but ART-tid and ART-tid provide an overall speedup.

Near-Data Processing

Architectural improvements for NDP have recently become an active research area [28], [5], [4], [11], [17], [18], [19], [20]. In NDP, processing power is brought closer to wherever data are stored in the memory hierarchy. Studies show that enhancing data storage with anything from lightweight compute power to a full ARM processor [16] leads to improved performance in nearly every tier of data storage. For example, new bit-line SRAM circuit technology enables Compute Caches [10] to compute in place, which nearly doubles system performance and cut

power usage in half. At the other end of the memory hierarchy, Riedel, et. al. introduced Active Disks [16] to enhance disk storage with software-downloadability so that application-level code can be run on directly on processors. Active-Routing [28] and this research propose techniques for augmenting the main memory system with processing power, an instance of the NDP paradigm called Processing-In-Memory (PIM).

The recent exponential growth in data availability in the big-data era and trends to move large amounts of data to memory [5] have sparked recent interest in the use of 3D-Memory technology for PIM. Several recent studies [5], [28], [4], [18], [21], [11] have made use of the HMC as the underlying 3D memory technology. For example, one of the first accelerators to make use of HMC technology was Tesseract, introduced by Ahn, et. al. [5], which includes hardware prefetchers specialized for parallel graph processing. Similarly, the Mondrian Data Engine [11] seeks to improve memory access patterns by embedding a processor, SIMD unit, object buffer, and stream buffer in each vault. Their algorithm-hardware co-design converts irregular memory accesses into sequential ones, despite the tendency for data analytics applications to partition data into different regions of memory.

Several other recent approaches to in-memory processing tend to focus on parallelizing computations [11], [5], [18], [4], [17], [30], [31], [18] which is the focus of this research. The Active Memory Cube (AMC) [17], [30] has been proposed as a way to integrate data-parallel applications by placing vector processing units in the logic layer of the HMC. Similarly, in [31], Fujiki, et. al. propose a programmable, in-memory processor architecture and a data-parallel framework that takes advantage of massive parallelism in TensorFlow inputs for machine learning. Their solution uses Non-Volatile Memories and takes advantage of native SIMD architectures in GPUs and SIMD Processing Units. In [18], Hong, et. al. seek parallelism through batching Linked-List Traversals (LLT). Their approach places an Engine driving LLT operations in each vault of an HMC. The system memory manger partitions memory into memory groups across the network so that nodes in the same linked list are often located in the same memory group. Thus, the approach only sees benefits from its parallelism if it partitions the memory in such a

way that several partitions can compute in memory at the same time, which only occurs when batching computations to memory. The approach leads to 2.4x throughput using batching, but the approach is not oblivious to the underlying data structure, while Active-Routing [28] targets general applications.

Computation in the Interconnection Network

Active-Routing [28] and other previous studies [23], [32], [24] have shown performance improvements when enhancing routing capabilities in the interconnection network. One of the earliest examples of processing in the interconnection network is the 1983 NYU Ultracomputer [23], which implemented adder logic in the switches of its interconnection network so that multiple fetch-and-add synchronization primitives could be coalesced in the network and generalized to other update operations. Later, in 1985, Pfister and Norton [32] proposed a technique for combining messages in multistage networks to avoid "Hot Spots," which can arise from situations where global shared locks degrade traffic in the network. Once packet-switching in NoCs became more common and sophisticated, Eicken, et. al. proposed Active Messages [24] to embed function pointers in messages to offload kernels to remote compute nodes. When one compute node needs to send large amounts of data to another, it embeds a pointer to a function that can load the data from the local memory, rather than communicating with the remote node. The approach saves power and significantly reduces overhead due to excessive data movement because the approach overlaps more computation with communication than previous approaches.

Several other recent proposals suggest improvements for MPI and Collective Communication [33], [34], [26], [25]. Many multi-node distributed systems implement the MPI Standard as a library for synchronization and other inter-process communications. The MPI Standard provides a rich library of message passing concepts, such as reductions over data from several processes and barrier synchronization. Two previous studys [25], [34] show that adding hardware support in the network and enhancing the network interface improves the performance of MPI collective communications, which are used for communication of shared data across processors, such as synchronization, broadcasting, or gathering of data. Ma et. al. [33] suggest that a Balanced, Adaptive

Multicast tree can collapse otherwise redundant reductions. In a multicast operation, messages are passed from one source to many destinations, but when all the receiving nodes reply with an ACK (acknowledgement) packet, the authors of [33] show that the messages along the same path can be synchronized and collapsed into a single message, which improves network saturation.

An increasing concern in data analytics is the use of machine learning, whose operations commonly include reductions over large sets of data (such as a dot product) and matrix multiplication. MAERI [27], an architecture recently proposed by Kwon, et. al., targets dataflow applications in Deep Neural Networks (DNN) with a topology that supports data-flow computations. The programmable DNN accelerator supports configurable DNN partitions, but the architecture still suffers from data movement from memory to SRAM and can only be computed in the leaves of the tree in the static topology. Active-Routing [28] supports topology-oblivious dynamic construction of such a tree and further minimizes data movement.

SECTION IV

COMPUTATION OFFLOADING TECHNIQUES

Through adjustments to the user-level API and enhancements to system configurations, we find that the baseline Active-Routing method can be further streamlined and that there is more opportunity for parallelism at the cube level. In this section, we describe our motivation for parallelizing computations in the vaults of an HMC by first describing the benefits of kernel offloading on the host CPU. Then, we explain the motivation for coalescing packets in the Memory Controller. Finally, we describe the benefits of combining requests for several cache lines in the same page of memory into a single `Update` packet.

Kernel Offloading

In the current implementation of Active-Routing, kernels are offloaded at instruction-level granularity [28], as they are done in several previous studies [4], [21], [18]. When a user writes a program using the API described in **Figure 7**, the compiler generates instructions for each `Update` to communicate with the Network Interface (NI), and when a processor reaches this instruction, it issues a request to the NI. The NI then sends the request to the Memory Controller through the NoC, and upon receipt of the request, the Memory Controller returns a response, allowing the processor to commit the `Update` instruction. Once the Memory Controller receives the request, it buffers the request until the associated entry port to the network becomes available, at which point the memory controller forms the `Update` packet and inserts it through the available port.

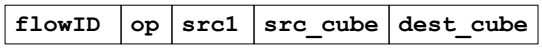
In typical systems with more cores than memory controllers, offloading instructions from cores to memory controllers has many-to-few (many cores to a few memory controllers) communication patterns. For example, in a mesh network as shown in **Figure 3**, the Memory Controllers only connect to the routers in the corners, then all of the other processors compete for bandwidth to reach the Memory Controllers. Such a communication pattern can lead to NoC saturation and thereby causing stalls in processor cores due to back pressure. Consequently, this bottleneck makes

`Update` packets sparse in the memory network, since the Memory Controller can only receive a few packets at a time from the NoC for offloading. To solve this problem, we offload the entire kernel to the Memory Controllers to bypass the NoC and to allow the Memory Controllers to generate Active-Routing packets locally. As a result, they can quickly offload packets into the memory network. This enhancement alleviates the need for additional hardware in the NI and improves the offloading throughput in the Memory Controllers. Each memory controller can be augmented with a simple SIMD instruction fetch/decode front end to facilitate offloading.

Update Packet Coalescing

Despite the benefits of offloading kernels to the Memory Controller, HMCs still sparsely receive packets due to contention for the limited bandwidth of links entering the memory network. In **Figure 4**, the Memory Controller can only communicate with the four corner HMCs. Each HMC still receives packets infrequently enough that the ALU remains largely underutilized. We observe that operand buffer usage does not significantly increase with kernel offloading, which means that `Update` packets are often consumed quickly because the queueing delay for the ALU is short. One reason `Update` packets arrive so sparsely to the HMC is because of the recurring overhead of sending several `Update` packets even they are addressed to the same cube.

To achieve higher utilization of the ALU and storage in the ARE, we propose to combine packets with the same metadata. When `Update` packets are generated in the Memory Controller, we propose coalescing transactions if the data are local to the same cube for the same flow. When generating a new packet, the Memory Controller can check if there are any outstanding transactions belonging to the same flow are destined for the same cube. For any two transactions the Memory Controller finds for the same flow, it first determines if the source data for that operand reside in the same cube. If they do, the packet can embed all the addresses of both transactions in a new transaction with the same metadata. For example, **Figure 10a** shows an `Update` transaction for a one-operand reduction before coalescing. **Figure 10b** shows the `Update` packet once N Transactions have been coalesced. In our experiments, we have seen only small numbers for N (10 or 12 coalesced transactions), so N does not require many bits of overhead in the new packet.



(a) Update Transaction Before Coalescing

(b) Update Packet Coalesced

Figure 10: Update Packet Coalescing. (a) Update Transaction for a single address before coalescing and (b) Coalesced packet with N source addresses

When the packet reaches its destination HMC, the source addresses are unpacked and sent to separate vaults at the same time. Thus, the link bandwidth becomes less of a limiting factor for offloading throughput. Therefore, the compute power in the ARE can achieve better effective throughput with more requests.

Page-Granular Offloading

Although packet coalescing in the Memory Controller improves offloading throughput, coalescing incurs significant overhead for the Memory Controller and ARE. The packet size grows significantly with increasing temporal and spatial locality (when coalesced transactions are both going to the same cube and issued at similar time). To amortize this overhead, we propose offloading operand data in page granularity. In [28], cacheline-granular offloading is achieved by specifying only the base address of the cache line in the packet because the entire cache block is guaranteed to reside in the same cube and the same vault. Similarly, if we allow packets to hold the base address of a page and specify the number of cachelines to be reduced, we can offload up to one page of computations to memory at a time. Such a way can better utilize the operand buffer and ALU in the ARE.

We demonstrate the use of this technique for a uniform reduction of a sum over a large set of data. We augment the API to make use of the function in **Figure 11** instead of consecutive calls to UpdateRR(). We then add logic in the Memory Controller to generate packets of this type. Finally, we enhance the Packet Processing Unit in the ARE to generate the appropriate operand requests based on the base address `src1` and range inside the page `num_lines`.

```
UpdatePage(void *src1, int num_lines, void *target, int op);
```

Figure 11: Update API for offloading consecutive cache lines in the same page

Because of the massive number of simultaneous requests made to different vaults of the same cube, data is often returned densely from the vault controllers to the ARE all at once. Therefore, the request rate is high enough to keep the ARE busy most of the time, leading to the maximum throughput of ARE. In many cases, it even introduces packet stalls on operand buffer resources because the compute throughput of a single ARE lags behind the data supply rate. This motivates vault-level parallelism to further increase compute throughput, which will be presented in the following section.

SECTION V

VAULT-LEVEL PARALLELISM

In this section, we describe our approach for enhancing Active-Routing with Vault-Level Parallelism (VLP). Specifically, we will first introduce the architecture and algorithm of this method using the previous ARTree example. Then, we will present our implementation of the approach. Finally, we discuss the different techniques employed for choosing a vault as the compute point in a cube.

Architecture

Vault-Level Parallelism makes use of the spare silicon in the HMC logic layer available for additional hardware near the vault controllers. Due to the distributed nature of the Active-Routing algorithm, inter-cube transactions are all treated the same way as in [28]. That is, the host CPU sees the same view of the HMC as in the previous section, and all architectural changes occur inside each HMC.

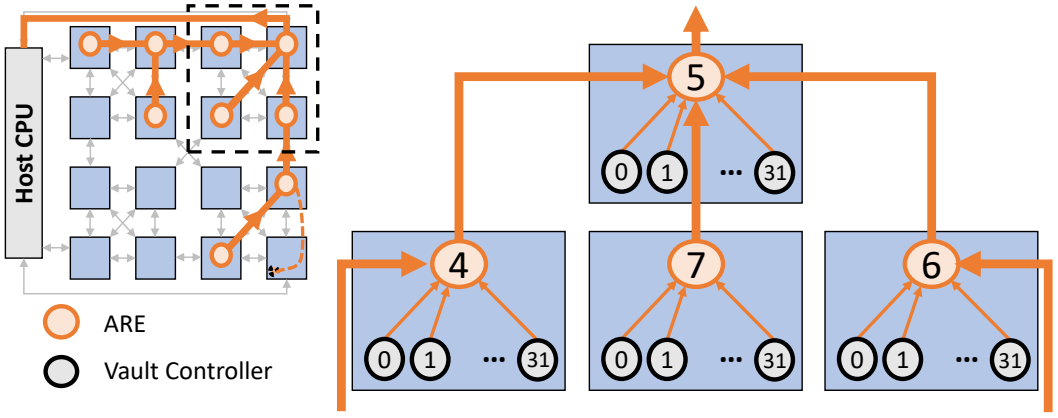
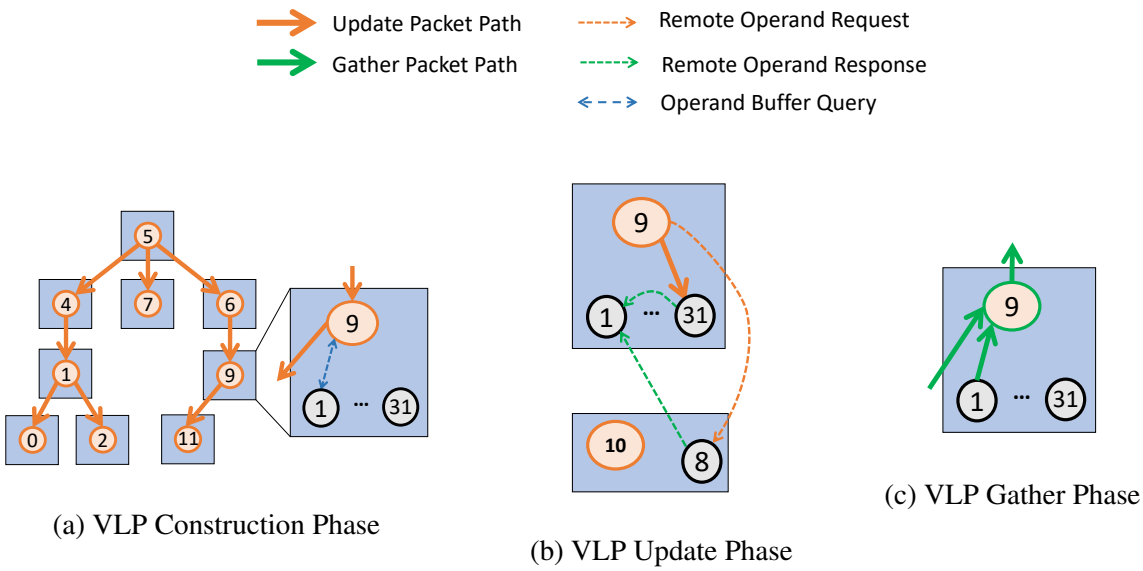


Figure 12: Role of Memory Cube and Vaults in Vault-Level Parallelism Architecture

VLP improves per-cube throughput by redistributing and adding resources in each vault

of the HMC. **Figure 12** shows the extended ARTree with VLP support. The ARE serves as a coordinator for the vaults and can be considered the root of a one-level ARTree where each vault involved in a flow is a child of the ARE. When Active-Routing packets are processed in the Packet Processing Unit of the ARE, the ARE forwards them to the appropriate vault and marks that vault as its "child" for that flow. Similarly, when a `Gather` packet is processed, the ARE distributes a copy to each vault involved in that flow and expects a response from each. Lastly, the ARE maintains a separate channel for communicating with each vault controller about operand buffer allocation. This channel will be discussed in detail later in this section.



(a) VLP Construction Phase

(b) VLP Update Phase

(c) VLP Gather Phase

Figure 13: 3-Phase Packet Processing for VLP extended ARTree for HMC 9

Figure 13 shows the inner workings of cube 9 during each phase of packet processing. During ARTree Construction (13a), the ARE uses its additional channel to record the flow information in vault 1. In the Update phase (13b), the ARE distributes `Update` packets to vault 31 in cube 9 and vault 8 in cube 10 (the remote request), encoding vault 1 of cube 9 as the compute point. The remote responses are sent back to cube 9 vault 1 for computation. In the Gather phase (13c), each vault embeds its local partial result in the `Gather` response packet and sends it back

to its parent ARE. Once the ARE receives all the `Gather` responses from its children, it initiates a `Gather` response towards its parent cube and commits the flow.

Implementation with An Example

Figure 14 shows the organization of the augmented HMC to support VLP. Starting with the Active-Routing implementation, each vault is granted a copy of all the ALU and Flow Table resources in the ARE. The operand buffers available to the ARE are then divided as evenly as possible among the vault controllers. Therefore, each vault controller has local compute capability and operand buffers so as to realize vault-parallel acceleration.

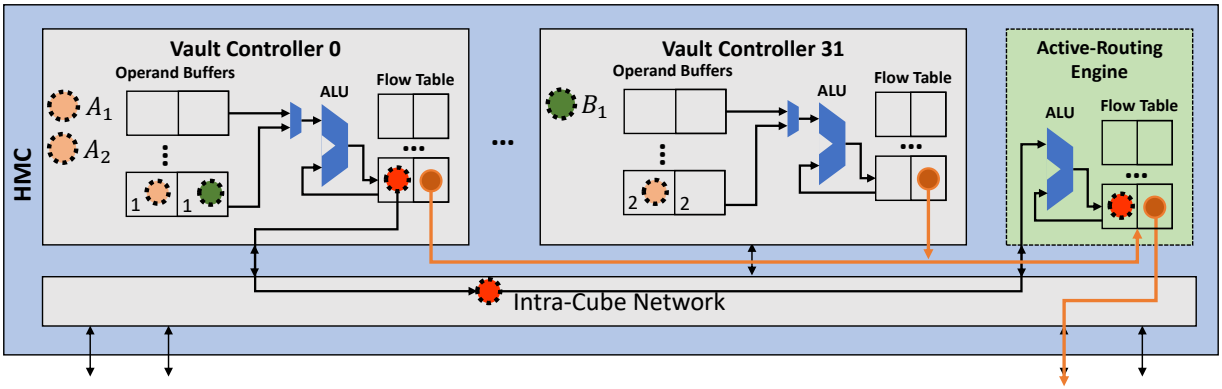


Figure 14: Vault-Level Parallelism Architecture for HMC 5 in **Figure 5**

Using **Figure 14** as an example, we demonstrate an example of VLP processing for HMC 5 of **Figure 5**. When an `Update` packet for $A_1 \times B_1$ first enters the HMC through one of the I/O links and the Packet Processing Unit unfolds the operands, the Flow Table is updated with an entry to record flow information. The ARE then inspects the packet and makes a decision about which vault to choose for offloading the computation. It makes use of its additional channel to each vault controller to query for operand buffer availability in the corresponding vault. If the ARE is not able to allocate an operand buffer to the `Update` packet, the ARE stalls the offloading of the computation until operand buffers are available. If a query to a vault results in a successful

reservation of an operand buffer, the ARE forms operand request packets for each of the operands in the same manner as in [28]. Each of these packets are then given an extra field indicating the vault in which the operand buffers were reserved. In the case of $A_1 \times B_1$, the ARE reserved an operand buffer in Vault Controller 0 and sent requests to vaults 0 and 31.

In vaults 0 and 31, the vault controllers record the compute vault of the outstanding request included in the packet meta data and generate DRAM commands to the DRAM command queue at the same time. When the data are returned from DRAM, the vault controller inspects the corresponding outstanding request of the response to determine whether it should send the response to another vault. In the case of A_1 in **Figure 14**, the data are immediately buffered, but in the case of B_1, the data are forwarded to vault 0 from vault 31. $A_1 \times B_1$ is then scheduled for computation in vault 0, and the result updates the partial reduction in the vault controller 0 Flow Table. In the case of $A_2 \times B_2$, the ARE follows the same procedure, as HMC is still chosen as the compute point in the network. One operand request is sent to vault 0 for A_2, and the request for B_2 is sent to the next cube where the data are located. When the data are available in vault 31 of HMC 5, they will be computed there.

Finally, when HMC receives a `Gather` packet from the HMC Controller, it copies the `Gather` packet for each of the child cube nodes in the ARTree. In addition, it makes another set of copies for the child vault nodes it has marked for the current flow. Similar to the Flow Tables of the ARE, the vault controllers mark their `gflag` field, return the `Gather` packet to the ARE to update the partial result kept in the Flow Table of the ARE, and commit the flow. When the ARE has received all the responses from the child vaults and child cubes, it responds to its parent cube in the ARTree with its partial result and commits the flow.

Dispatching Algorithms

One design choice that affects the performance of VLP is the choice of which vault to use as the compute point. To avoid the additional overhead of response messages for inter-vault communication as shown in **Figure 13b**, the algorithm for dispatching `Update` computations to vaults should be carefully designed. We propose two methods for choosing a compute vault in a

given HMC. We leave further optimization of this dispatching algorithm for future work.

The first approach is to dispatch computations to all the vaults in a round-robin manner. For each new incoming `Update` packet, statically choose the next vault without inspecting the packet contents. Although this approach has the potential to lead to many remote operand requests, it utilizes all the operand buffers in the cube. Unfortunately, the delay and overhead of remote operand requests often ruins performance before operand buffers are saturated, especially for computations that requires two source operands.

The second approach, namely content-aware, suggests inspecting the `Update` packet before dispatching to vaults. If the packet has any operand(s) in the current cube, the ARE tries to reserve operand buffers and schedule it to compute at the local vault of the operand first. If those vaults do not have free operand buffers, the ARE defaults to the static round-robin and avoids querying the vaults that have already been queried. This approach leads to much better performance but often at the cost of storage. When distributing the operand buffers to each of the vaults, each HMC often gets very few operand buffers. If one vault is a "hot spot" for computations (i.e. it contains much of the data needed for that flow), it will quickly run out of operand buffers and require the use of operand buffers from other vaults. With room to expand the hardware budget, the content-aware technique performs much better for kernels with high data locality.

SECTION VI

METHODOLOGY

In this section, we describe our system configuration and modeling tools we used to simulate the techniques we implemented, as well as the workloads we used to test the performance of our techniques.

System Configuration and Simulation

McSimA+ [35] simulates the microarchitecture of the cores and cache hierarchy in the backend of our simulations using an execution-driven environment. We use CasHMC [36] for cycle-accurate simulation of the HMC memories. To implement Active-Routing instructions, such as those generated for UpdatePage(), we use the tool Pin [37] supported by McSimA+'s frontend. We model the Active-Routing Engine logic in the Crossbar Switch and implement the vaut-level Active-Routing logic directly in the vault controller.

We describe our system configurations in **Table 1**, as is depicted in **Figure 4**. We show here the difference in design hardware between the baseline configuration and the VLP configuration for the ARE and the vault controllers. We also configure the host CPU as a CMP with on-chip network and a two-level cache hierarchy wit MESI coherence protocol. The memory network configuration is a Dragonfly topology, as adopted from [12] in [28].

Workloads

Vault-Level Parallelism targets the same applications as [28]. Vault-Level Parallelism provides the most benefit to Active-Routing when reductions on large sets of data involve data-flow with the opportunity for massive amounts of parallelism. In this work, we focus mainly on pure reductions and multiply-and-accumulate operations. We evaluate the performance of our kernel offloading techniques, packet coalescing, page-level offloading, and the VLP system configuration using microbenchmarks available from [28] and kernels from benchmark suites that are use for

Table 1: System Configurations

Parameter		Baseline Configuration	VLP Configuration
CPU	Core	16 OoO cores @ 2GHz issues/commit width: 4, ROB: 128	
	L1 I/D Cache	Private, 32 KB, 4-way	
	L2 Cache	S-NUCA, 16 MB, 16-way, MESI	
	NoC	4x4 mesh, 4 MC at 4 corners	
Memory	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	0 buffer entries
	Processing Element	1250 MHz clock frequency An arithmetic logic unit	
Vault Controller	Flow Table	NA	16 flow entries
	Operand Buffer	NA	4 buffer entries
	Processing Element	NA	1250 MHz clock frequency An arithmetic logic unit

scientific computing. We re-implement these kernels with the Pthread library to maintain compatibility with McSimA+ and use a sufficiently large data size to stress the last-level cache, memory controller, and memory network. We summarize these workloads in **Table 2**.

Table 2: Evaluation Workloads

Workloads		Optimization Region	input Data Size
Benchmarks	sgemm [38]	matrix multiplication	4096x4096 matrix
Microbenchmarks	reduce	sum reduction over a sequential vector	6400K dimension
	mac	multiply-and-accumulate over two sequential vectors	two vectors with 6400K dimension

SECTION VII

RESULTS

In this section, we first present our experimental results from simulations of our offloading techniques. Then, we compare performance between baseline and VLP architectures followed by the performance results of round-robin and content-aware dispatching techniques. Finally, we demonstrate the results of sensitivity testing between VLP and baseline architectures.

We use the Instruction Offloading (IO) and Kernel Offloading (KO) techniques discussed in the previous section as suffixes to the names of each architecture. Baseline refers to the Instruction Offloading architecture supported in [28] with none of the offloading techniques introduced in this research. Architectures that support Page-granular Update packets are then further specified with Page. Architectures supporting Update coalescing are specified with Coalesce. For example, a VLP architecture that supports Kernel Offloading and Page-granular offloading would be referred to as VLP-KO-Page. All the technique combinations are summarized in **Table 3**.

Table 3: Evaluation Workloads

Techniques	**Description**
Baseline	Instruction Offloading Active-Routing [28]
Kernel Offloading (KO)	Compute kernel offloaded to Memory Controller for Active-Routing
Coalesce-KO	Coalescing packets of same flows that go to the same cube with Kernel Offloading
IO-Page	Page-granular `Update` packet using Instruction Offloading
KO-Page	Page-granular `Update` packet using Kernel Offloading
VLP-IO	Vault-level parallelism support with Instruction Offloading
VLP-KO	Vault-level parallelism support with Kernel Offloading
VLP-IO-Page	VLP-IO with Page-granular `Update` packet
VLP-KO-Page	VLP-KO with Page-granular `Update` packet

Offloading Optimizations

Figure 15 compares the runtimes resulting from the different offloading techniques introduced in Section IV. We show the normalized runtime speedup over Baseline for microbenchmarks *mac* and *reduce* as well as the *sgemm* (dense matrix multiplication) benchmark. The results demonstrate that more frequent delivery of Update packets to HMCs significantly cuts down on runtime, but coalescing Updates in the Memory Controller does not improve performance much. We observe that there is no stalls in the ARE, meaning the request rate does not keep up with computation throughput, leaving compute units underutilized. Next, we turn to page-level granular Update packets to offload more computations to the Memory Controller with less overhead.

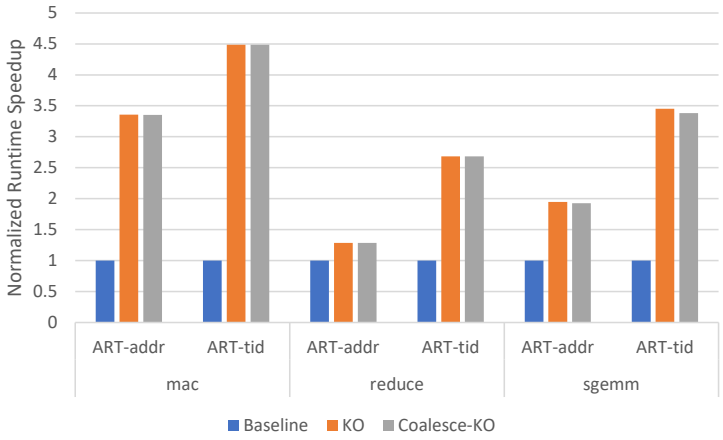


Figure 15: Runtime Speedup for all workloads using Kernel Offloading and Update coalescing

Our technique for page-granular Update packet offloading can be demonstrated on any operation. In this work, we show the results of implementing the architecture for a pure reduction as a case study. Since page-granular Update packets can offload up to a whole page in a single instruction, which significantly amortizes the overhead of offloading, thereby effectively increasing offloading throughput. We present page-granular offloading both with and without the kernel offloading technique, and compare the runtime speedup over kernel offloading on *reduce* as shown

in **Figure 16**.

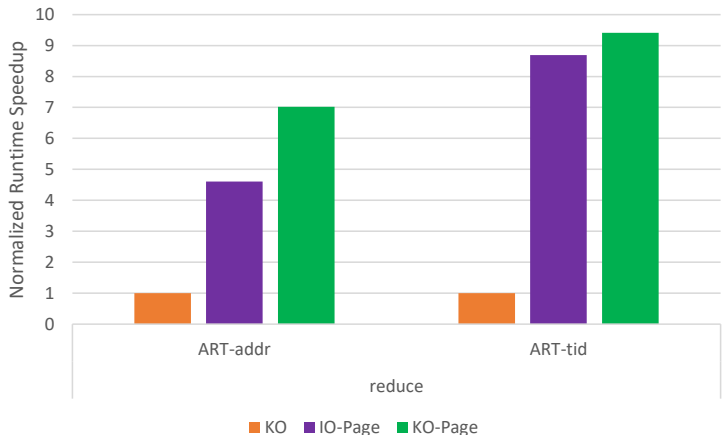


Figure 16: Runtime Speedup of reduce using page-granular Update packets

With page-level granular instruction offloading, we observe a $8.6\times$ speedup over kernel offloading. When page-level granularity combines with kernel offloading, it achieves $9.4\times$ performance improvement. Regardless of such significant improvements, we notice that the operand buffers in the ARE stall significantly more often when page-granular offloading is applied. This is because the compute throughput does not keep up with the data supply rate. Therefore, we turn to parallelizing these operations in the vaults while keeping the hardware budget for operand buffers the same.

Vault-Level Parallelism

Figure 17 shows the runtime speedup of VLP with and without kernel offloading compared with the instruction offloading Baseline. We again compare runtimes normalized to the kernel offloading and page-granular version of the approach in the previous subsection, which we refer to as baseline for this subsection. For all the experiments in this subsection, we use the round-robin vault dispatching policy. Notably, the parallelism available in VLP does not make up for the low throughput of Update packets in the Memory Controller when kernel offloading is not enabled.

Even with kernel offloading enabled, however, VLP tends to match or perform slightly worse than the baseline implementation with kernel offloading. We now turn to page-level granular offloading to make use of the parallel compute power available in the vaults.

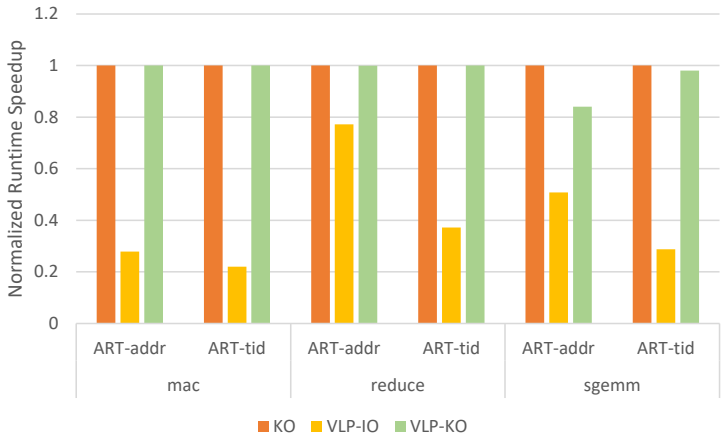


Figure 17: Contribution of Kernel Offloading to workload performance on VLP Architecture

Page-level granular offloading, shown in **Figure 18**, utilizes the parallelism available in each cube. In this figure, we show the speedup of the VLP architecture over the version of the original architecture with both kernel offloading and page-granular Update packets to demonstrate the increased throughput0 parallelism contributes to the system. Even without kernal offloading, the approach still performs reduce about $1.5\times$ faster than the architecture proposed in [28]. The increased throughput from the parallel nature of the architecture allows fewer stalls due to operand buffer contention between Updates which are present in the original architecture.

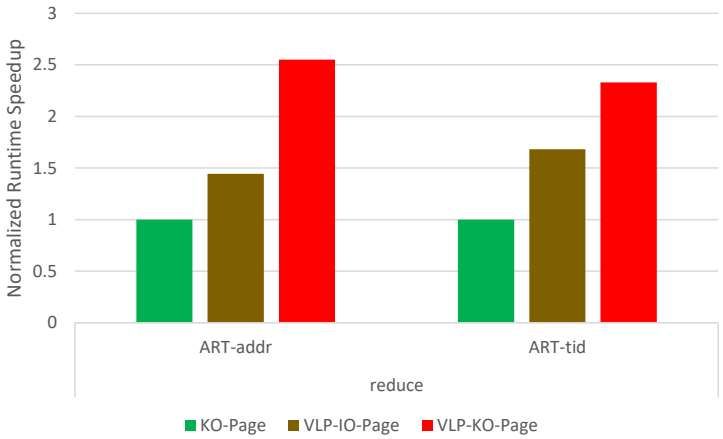


Figure 18: Runtime Speedup of reduce enabled by page-granular Updates in the VLP architecture

Dispatching Algorithms

We experimented with two classes of dispatching procedures (Round Robin and Content-Aware) in the ARE that warrant attention for future research in Vault-Level Parallelism. While Round Robin statically assigns a vault as the first vault for dispatching and always queries the next vault in sequence by vault ID, the Content-Aware approach inspects the packet to determine where the operand(s) are coming from. Our implementations for all the previously mentioned results involve the use of Round Robin starting from the vault with the lowest ID. In this subsection, we explore some other techniques for compute dispatching in the cube for two-operand cases. For one-operand cases, content aware will always perform the best, but in the case where two operands do not reside in the same cube, we observe similar performance for both dispatching algorithms. We only analyze dispatching techniques of VLP-KO in this thesis. In the future, we plan to study for Page-granular offloading as well as scheduling algorithms for determining compute cubes. We demonstrate the Round Robin and Content-aware approaches for different numbers of operand buffers in **Figure 19**.

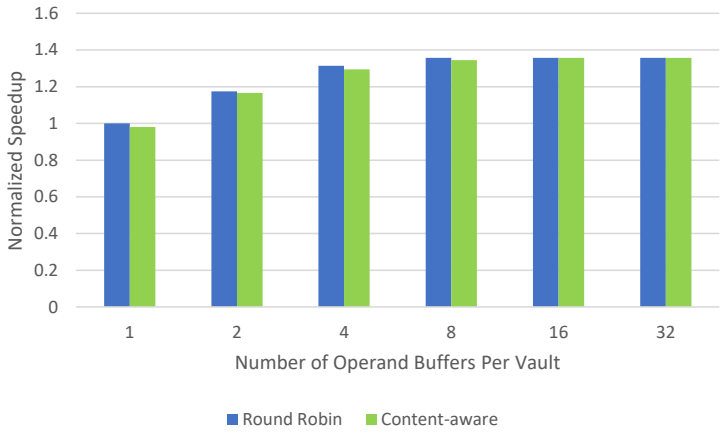


Figure 19: Runtime Speedup of sgemm Benchmark over different operand buffer budgets

We also evaluate the sensitivity of VLP to the number of operand buffers by varying the size as shown in **Figure 19**. For both approaches, fewer operand buffers means more stall time when the operand buffers become full in the whole cube. After the number of operand buffers per vault is more than 4, the performance improvement becomes very marginal. When the number of operand buffers is more than 8, the performance is almost same. Therefore, we choose 4 as the default operand buffer size for each vault.

SECTION VIII

CONCLUSIONS

Active-Routing performance is restricted both by the costly instruction offloading and limited compute power. This research introduces two proposals to combat these problems. We first investigate several methodologies for reducing the overhead of instruction offloading inherent in [28], and then we propose a new architecture that uses the spare silicon in the HMC logic layer to scale to the increased throughput demands. We first propose kernel offloading as a way to bypass the contention for links in the NoC among processors. Although kernel offloading techniques increase the throughput of packets into the memory network, the offloading throughput is still limited by the small number of operand addresses that can be packed. We introduce a new API for offloading an entire page of memory with a base address and number of cache lines. This page-granular offloading allows each ARE to unfold dozens more operands in each packet without increasing the size of the packet, which saturates the compute throughput of the HMC. Second, we propose the Vault-Level Parallelism Architecture, which treats each ARE as the parent of a reduction branch in each HMC. Using the spare silicon in the HMC logic layer, vault controllers are augmented with their own copy of Active-Routing logic for storing the state of flows, decoding packets, and computing partial results. We study the effects of this parallel architecture in a case study using a one-operand pure reduction. Our simulations show a $23\times$ speedup over the original baseline in [28] when enhancing VLP with kernel offloading and page-granular offloading.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc, 2017.

[2] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, pp. 1–24, Aug 2011.

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

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

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 105–117, June 2015.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[7] T. Suzumura, S. Nishii, and M. Ganse, "Towards Large-Scale Graph Stream Processing Platform," in *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, (New York, NY, USA), p. 1321–1326, Association for Computing Machinery, 2014.

[8] D. S. Modha, R. Ananthanarayanan, S. K. Esser, A. Ndirango, A. J. Sherbondy, and R. Singh, "Cognitive Computing," *Commun. ACM*, vol. 54, p. 62–71, Aug. 2011.

[9] 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 *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 190–200, March 2014.

[10] 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)*, pp. 481–492, Feb 2017.

[11] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian Data Engine," *SIGARCH Comput. Archit. News*, vol. 45, p. 639–651, June 2017.

[12] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with Hybrid Memory Cubes," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pp. 145–155, Sep. 2013.

[13] K. Kondo, M. Kada, and K. Takahashi, *Three-Dimensional Integration of Semiconductors: Processing, Materials, and Applications*. Springer International Publishing, 2015.

[14] 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, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, "25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 432–433, Feb 2014.

[15] 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 *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–14, 2016.

[16] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle, "Active disks for large-scale data processing," *Computer*, vol. 34, pp. 68–74, June 2001.

[17] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien, and et al., "Data Access Optimization in a Processing-in-Memory System," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[18] B. Hong, G. Kim, J. H. Ahn, Y. Kwon, H. Kim, and J. Kim, "Accelerating linked-list traversal through near-data processing," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 113–124, Sep. 2016.

[19] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules,"

in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 283–295, 2015.

[20] K. Hsieh, E. Ebrahim, 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 *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 204–216, 2016.

[21] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), p. 336–348, Association for Computing Machinery, 2015.

[22] J. Ahn, S. Yoo, and K. Choi, "AIM: Energy-Efficient Aggregation Inside the Memory Hierarchy," *ACM Trans. Archit. Code Optim.*, vol. 13, Oct. 2016.

[23] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—Designing a MIMD, Shared-Memory Parallel Machine," in *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, (New York, NY, USA), p. 239–254, Association for Computing Machinery, 1998.

[24] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, pp. 256–266, 1992.

[25] D. K. Panda, "Global reduction in wormhole k-ary n-cube networks with multidestination exchange worms," in *Proceedings of 9th International Parallel Processing Symposium*, pp. 652–659, April 1995.

[26] 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 *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–10, Nov 2011.

[27] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects," *SIGPLAN Not.*, vol. 53, p. 461–475, Mar. 2018.

[28] 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," 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), p. 674–686, IEEE, Feb 2019.

[29] N. E. Jerger, T. Krishna, L.-S. Peh, and M. Martonosi, "On-Chip Networks, Second Edition," vol. 12, no. 3, p. 1–210, 2017.

[30] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. . 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, pp. 17:1–17:14, 2015.

[31] D. Fujiki, S. Mahlke, and R. Das, "In-Memory Data Parallel Processor," *SIGPLAN Not.*, vol. 53, p. 1–14, Mar. 2018.

[32] 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.

[33] S. Ma, N. E. Jerger, and Z. Wang, "Supporting efficient collective communication in NoCs," in *IEEE International Symposium on High-Performance Comp Architecture*, pp. 1–12, 2012.

[34] D. K. Panda, "Global reduction in wormhole k-ary n-cube networks with multidestination exchange worms," in *Proceedings of 9th International Parallel Processing Symposium*, pp. 652–659, April 1995.

[35] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 74–85, 2013.

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

[37] 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," *SIGPLAN Not.*, vol. 40, p. 190–200, June 2005.

[38] J. A. Stratton, C. I. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," 2012.