ACTIVE ROUTING: COMPUTE ON THE WAY FOR NEAR-DATA PROCESSING

A Thesis

by

RAMPRAKASH REDDY PULI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2018

Major Subject: Computer Engineering

ABSTRACT

The explosion of data availability and fast data analytic requirements led to the advent of data-intensive applications, characterized by their large memory footprint and low data reuse rate. These data intensive applications place a significant amount of stress on modern memory systems and communication infrastructure. These workloads, ranging from data analytics to machine learning, exhibit a considerable number of aggregation operations over large data sets, whose performance is limited by the memory stalls due to widening gap between high CPU compute density and deficient memory bandwidth.

This work presents Active-Routing, an In-Network Compute Architecture enabling compute on the way for Near-Data Processing (NDP), which reduces data movements across the memory hierarchy. It moves computations close to data location onto the memory network switches which operate concurrently and construct an Active-Routing tree. The network is enabled with computing capability to optimize the aggregation operations on a dynamically built routing tree to reduce network traffic and parallelize computation across the memory network. Evaluations in this work show that Active-Routing can achieve up to 6x speedup with an average of 75% performance improvement across various benchmarks compared to a Baseline system integrated with a die-stacked memory network.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Professor Dr. EJ Kim of the Department of Computer Science Engineering and Professors Dr. Jiang Hu and Dr. Gwan Choi of the Department of Electrical and Computer Engineering.

The idea presented in Chapter III was formalized in part by Jiayi Huang and Pritam Majumder of the Department of Computer Science and Engineering. The simulator infrastructure and analysis tools used in Chapter IV were developed in part by Jiayi Huang of the Department of Computer Science and Engineering.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was supported by a Graduate Merit Scholarship from the Department of Electrical and Computer Engineering, Texas A&M University.

NOMENCLATURE

ISA	Instruction Set Architecture
NDP	Near Data Processing
PIM	Processing-In-Memory
CPU	Central Processing Unit
CMP	Chip Multiprocessor
HMC	Hybrid Memory Cube
TSV	Through-Silicon Via
ILP	Instruction Level Parallelism
TLP	Thread Level Parallelism
MLP	Memory Level Parallelism
AR	Active Routing
ALU	Arithmetic Logic Unit
ARE	Active Routing Engine
API	Application Interface
MI	Message Interface
ROB	Reorder Buffer
NUCA	Non-Uniform Cache Architecture
EDP	Energy-Delay Product

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TABLE Page

1. INTRODUCTION

With the onset of big data era, modern data-intensive applications require a large amount of on-off chip data movements across the memory hierarchy due to limited cache sizes and irregular memory access patterns. This has further worsened the issue of widening gap between computation speed and memory system performance, namely the memory wall [1]. As a result, modern large-scale systems fail to achieve their peak computational capability for data-intensive applications. Therefore, it is critical to find solutions that can circumvent the memory wall.

Near-Data Processing (NDP) or Processing-In-Memory (PIM) [2, 3, 4], a paradigm to reduce data movements across the memory hierarchy by moving computation near to data location was introduced in the 1900s [5]. PIM has gained significant amount of interest recently due to the modern technological advancements like 3D Die-Stacking [6] such as in Hybrid Memory Cubes (HMC) [7] facilitating compute units close to memory dies. The need for higher performance and larger memory bandwidth/capacity has driven processor architectures to adopt multi-core designs with more than one memory slot on a single chip. The number of such memory slots that can be placed on a chip is limited by the physical chip-pin count. Interconnection networks have been the ubiquitous solution for communication infrastructure among such rapidly scaling processing/memory systems. Recent works have proposed memory-centric network designs to interconnect memory modules as a network to increase memory capacity by overcoming the chip-pin bottleneck and fully utilize the processor and memory bandwidth [8, 9]. These designs use passive network switches to route memory requests/responses over a scalable fabric of memory modules.

However, with NDP/PIM the current passive communication networks are still not optimized to reduce the network traffic for different tasks in data-intensive applications. Though NDP/PIM systems can reduce the data movement to the compute nodes, they still incur the costs of transferring remote source data or gathering the compute results. For example, a multiply-accumulate operation $sum \mathrel{+}= input[i] * weight[j]$ in deep learning applications will require the movement of all the elements of the arrays input and weight to either the compute node or to the location of target data sum in the memory. With big data applications from graph analytics to deep learning, the intensive use of aggregation on another operator over a large data set puts high pressure on memory system and communication infrastructure. Therefore, it is critical to design an efficient memory network architecture which optimizes network traffic due to reduction/aggregation operations on large data sets for better system performance.

This work proposes a novel interconnection network solution, Active-Routing, to comprehensively optimize data processing phase as well as reduction operations as a whole. The main idea boils down to moving the computation closest compute resource to the location of source operands in memory network and aggregate the results in the network along a dynamically built routing tree. As an example, the operation $sum \mathrel{+}= input[i] * weight[j]$ is optimized by building a dynamic tree with nodes as memory modules in the memory network where optimal nodes (in the tree) are identified, on an individual basis of each (i, j), to compute the summation of each of the corresponding multiplication result of data arrays input and weight. The optimal memory node in such a dynamically built routing tree is the common ancestor of the nodes where the operands are located. Active-Routing is seamlessly integrated with current systems using instruction set extensions to offload special load/store Update and Gather instructions through message interface. A programming interface is provided to convert the program semantics into these instruction set extensions. It offloads operations as PIM computations through Update packets to dynamically build Active-Routing trees on-the-fly, concurrently it initiates data processing when operands are ready on memory network switches. Reduction operations over results of Update instructions belonging to the same memory node are processed as soon as they commit. A following Gather packet handles the reduction along the routing tree to obtain the final aggregated result.

The main advantages of Active-Routing are: (a) it reduces data movement for data-intensive applications especially pointer referenced accessing; (b) data processing happens in concurrent with task offloading with minimized stalls; (c) it eliminates the coherence communication overheads of shared data update; and (d) reduction/aggregation can be performed along the routing path to reduce the network traffic and latency. Evaluations from this work show that *Active-Routing* can

achieve upto 6x speedup with a geomean of 75% performance improvement across various real world benchmarks compared to a Baseline system integrated with a die-stacked memory. It is also observed that *Active-Routing* achieves high efficiency with an average Energy-Delay Product (EDP) improvement of 88%.

2. BACKGROUND

Die-Stacked Memory [7, 6] architectures possess some underlying characteristics, which can be leveraged to improve the system performance through NDP. This chapter introduces such characteristics and later discusses the data movement bottleneck of data intensive applications and inherent limitations of synchronization constructs in multithreaded programs, which motivates *Active-Routing*. Later, three directions related to memory and memory-network research are discussed to further ascertain the need for this work.

2.1 Die-Stacked Memory

Advancements in memory technology have facilitated the integration of logic and memory dies in the same package using 3D Die-Stacking. The DRAM stacks are connected to the logic die with low-latency, high bandwidth through-silicon vias (TSVs) etched vertically through the stacked dies [6]. Hybrid Memory Cube (HMC) [7] is one popular example of die-stacked DRAM. HMC is vertically partitioned into several functionally independent vaults consisting of DRAM banks and multiple TSV connections. Each vault has a memory controller called a vault controller, placed on the logic layer that manages all memory references within that vault. These vault controllers are sparsely spaced leaving ample amount of unused silicon area on the logic layer [3]. Figure 2.1 shows an HMC organization with 8 memory and a logic layer consisting of vault controllers. HMC provides high speed Ser/Deser links to connect it to processor or other HMCs, supporting packetized communication among them. A crossbar switch is located on the logic layer which routes packets to their destination vaults.

There are several benefits of die-stacked DRAM over the conventional DRAM memory. For example, HMC provides better scalability in terms of memory capacity and bandwidth. Die-stacking technology enables larger memory size per package while fast TSVs and Ser/Deser link protocol leads to abundant internal and external bandwidth. The packet-based communication protocol gives flexibility in extending the type of commands supported. These advantages are leveraged in

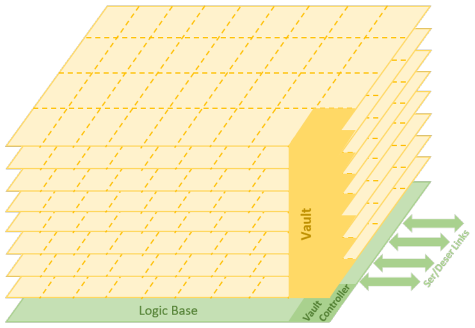


Figure 2.1: Hybrid Memory Cube Organization

many existing NDP and PIM studies [2, 10, 11, 4, 3]. The logic layer is leveraged to implement computation capability ranging from limited fixed functionality [2, 10, 11] to full-fledged in-order processors [3]. Existing research has shown significant improvements of NDP/PIM paradigm over the conventional systems by offloading computation to the memory. This work exploits NDP paradigm not only to offload operations to memory, but also to enable routers with active functionality to compute while transferring data on the route.

2.2 Memory Network

Conventional systems with DRAM memory have capacity limit and bandwidth bottleneck due to limited number of physical pins of the processor chip. Therefore, scaling of memory capacity in traditional systems require more processor sockets to connect DRAM packages. However, in data intensive applications, data movement and memory requirement is much higher than computation. This stalls compute units for the availability of data operands and leads to underutilization of computation resources. In contrast, HMCs can be chained together to form a pool of memory or a memory network for providing larger memory capacity as necessary. In addition, current system designs mostly use processor-centric interconnection networks to optimize processor-processor communication but overlook the possibility of communication among memory cubes. Recent studies [9, 8] have shown that the memory-centric interconnect design can provide better bandwidth utilization as compared to the processor-centric design for traditional computation paradigm. In such memory-centric interconnect system memory modules such as HMCs are interconnected in various topologies and memory requests and responses are routed ideally utilizing both processor interconnect and the memory interconnect. *Active-Routing* exploits NDP to embed computation of data while transferring on the route during communication from data location in memory to processor. Therefore, this work adopts a unified memory-centric network [9] design to provide better bandwidth as well as a better fit for *Active-Routing*.

2.3 Motivation

The explosion of digital data and fast data analytic requirements lead to the rise of emerging data-intensive applications, which have very low operations per data and large data movement demand [12, 13]. The low data reuse rate of large working sets can cause high cache miss rate and incur low energy efficiency of data movement from or to memory. Furthermore, such characteristics can force the processor to be stalled for the data to arrive from memory and hinder the thread-level parallelism (TLP) as well as instruction-level parallelism (ILP). To scale the performance with the data generation, *Active-Routing* moves computation from processor to memory instead of moving data from memory to processor through NDP. With NDP, we extract parallelism from multi-threaded applications to exploit memory-level parallelism (MLP) of die-stacked memory and network concurrency.

Multi-threaded programs achieve good performance by exploiting TLP in the CMP environment. However, mostly the benefits of TLP are partially offset by the synchronization constructs, because of inherent limitations in hardware to maintain memory consistency. In conventional systems it is hard to preserve the program semantics of multi-threaded programs due to race conditions without imposing constraints on execution. Hence, synchronization constructs such as barriers/locks are used to synchronize threads at certain points in program where they need to operate on some shared variables. This kind of operations limit the performance of applications to a good extent and are unavoidable. A subset of this problem can be solved by providing the constant operand along with the memory request and operate them atomically in the memory [2]. However, such an operation still requires to bring one or more source operands to the processor side, which may offset the benefit of offloading to compute in memory. *Active-Routing* first offloads operations near memory to reduce data movement of source operands and builds an *active-routing tree* dynamically. Then the router switches are upgraded with computation capability to aggregate the commutative computed results on the way while transferring the partial results along the *active-routing tree*.

2.4 Related Work

The idea presented in this work is at the intersection of various research directions in high performance computer architecture designs to improve system performance. This section discusses three such directions and presents some previous works to distinguish them from *Active-Routing*.

2.4.1 Near-Data Processing

Paradigms like NDP have been proposed a long time back [5] to reduce the overhead of data movement across the memory hierarchy, but did not come to fruition because of the limitations in the memory technology at that time. These paradigms are currently experiencing a resurgence in interest [2, 3, 4, 14, 11, 15, 10, 16] from architecture community, driven by the advances in memory technology [6, 7] like 3D Die-Stacking, and the increasing memory bandwidth stress imposed by emerging big data applications [12, 13]. Ahn et al. [2] proposed to monitor data locality and process the fetch-and-update operations either on host or offload it via extended instructions to memory. This mechanism limits to fetch-and-update operations which only carries the memory target, and also the locality monitoring incurs back-invalidation and back-updates for coherence. Ahn et al. also proposed Tesseract [3], a programmable PIM accelerator for large-scale graph processing. Tesseract accelerates data intesive kernels by offloading them entirely on to in-order processor cores embedded on memory. Nair et al. [14, 15] proposed a new architecture, Active Memory Cube (AMC), to perform computation in the memory module for exascale systems. They propose to leverage the commercially demonstrated Hybrid Memory Cube (HMC) to place vector architecture based computation elements (integer, floating-point, predicate and scatter-gather) in the logic layer. Their mechanism suffers from requirements to pre-load instructions into the Instruction buffer of these vector compute elements by the host, address translation from the host and a complex interconnect for fully connecting vaults and lanes in each quadrant of the AMC layout.

2.4.2 Leveraging Commutativity in Operations

There has been considerable research exploring mechanisms to leverage the commutativity/associativity in operations to improve the effective performance by alleviating the overheads of coherence or gathering [17, 4]. AIM [4] proposed to reduce the data movement on-off chip by performing the commutative and associative aggregation at the data location without modifying the coherence protocol, thereby improving performance and saving energy. Zhang et al. [17] presented COUP which augments the cache coherence protocol so that the several copies of a shared variable can be updated in parallel. It is especially beneficial for applications with finely interleaved reads and updates to shared data. Both AIM and COUP have to bring source operands to CPU for updates while *Active-Routing* can leverage NDP for source data as well.

2.4.3 Processing in the Interconnection Network

Previous research [18, 19, 20] has encouraged interconnection network to offer more functionalities other than just routing packets. Active Message [18] embeds the function pointer and arguments across the network to perform tasks in remote compute nodes. Pfister et al. [19] and Ma [21] proposed mechanisms to combine messages so as to reduce network traffic. Recently, IncBricks [20] implements an in-network caching middlebox for key-value acceleration in router switches. Several studies [22, 23, 24] proposed mechanisms to optimize shared value update or reduction in the network. The NYU Ultracomputer [22] implemented adder in network switches to coalesce the atomic fetch-and-update for same target address along their way to memory. Panda [23] and Chen et al. [24] describe similar mechanisms that provides network interface functionality as well as hardware support for MPI collective reduction communication in a static manner. These mechanisms, targeting at shared variable update optimization, cannot be applied to optimize data processing phase before reduction, thus requiring large amount of data movement for the source data from memory to CPU. *Active-Routing*, on the other hand supports data processing in the memory network as well as reduction on the way along a *topology-oblivious* dynamically built *Active-Routing tree*.

3. ACTIVE ROUTING ARCHITECTURE

This chapter describes the system and communication architecture and then detail the components that implement Active-Routing. Figure 3.1 presents the system configuration where host processors are connected to a memory network of Hybrid Memory Cubes (HMC) [7]. Processing cores on the host side and the cache hierarchy communicate with each other through an on-chip network. On the other hand, processing cores communicate with the memory subsystem via HMC controllers connected to a high bandwidth memory network. Each memory module is enabled with router switches, built into the logic layer of the memory cube to establish communication among the cubes.

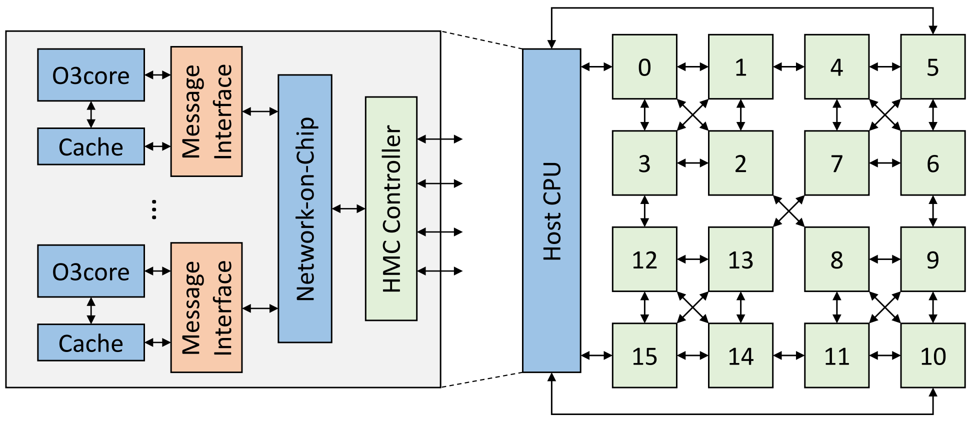


Figure 3.1: System Configuration with a Host CMP Connected to a Memory Network.

Active-Routing is an active interconnection network processing solution to optimize reduction/aggregation operations, with multiple operands. The main idea is as follows:

- Computation is moved to the memory cube to facilitate data processing on the memory side. Hence, mitigate the data supply overhead across deep memory hierarchy.
- Topology-oblivious Active-Routing tree is constructed dynamically in parallel with the computation offloading and data processing in the memory network.

- Reduction operations are optimized by performing partial result reduction in the network along the Active-Routing tree (ARTree) from leaf nodes to the root.

There are multiple components which work in synergy to realize Active-Routing. Primarily an Application Interface (API) is required to facilitate offloading computation to data-resident locations and perform seamless reduction operations along the ARTree. Simple programming interface (Update and Gather), is provided to translate operation semantics into extended instructions for offloading data processing and reduction operations. The programming interface is discussed in detail in Section 3.1. Additionally, microarchitectural modifications across the on-chip and memory networks are necessary to support Active-Routing functionalities. An Active-Routing Engine (ARE) is added to the router switches of the memory network to implement the functionalities. An ARE consists of an ALU for near data computation, a Flow Table to track multiple concurrent active flows, a pool of operand buffers to store data processing operands, and a packet decoder for the Update and Gather traffic. Section 3.2 and 3.3 present the detailed components of ARE and their functionalities in realizing *Active-Routing*. Later in Section 3.4, design considerations to seamlessly integrate this architecture into current systems are discussed.

3.1 Programming Interface and ISA Extension

A simple programming interface, (**Update** and **Gather**) is provided to translate the operation semantics into extended instructions. The ISA extensions are used to communicate with the Message Interface to offload the computation to data-resident locations for NDP as well as perform result reduction along the *ARTree*.

3.1.1 Programming Interface

```
Update(void *src1, void *src2, void *target, int op);
Gather(void *target, int num_threads);
```

The above **Update** API carries two source memory addresses for processing, and a target address to register a unique flow entry in the Flow Table along the way to the near-data location. The `op` parameter indicates the operation to be done in place (e.g. multiplication) through **Update** command packets. The `op` can be used as an opcode and the memory addresses can be conveyed to the Message Interface through registers. Similarly, the **Gather** API embeds the target address as a flow identification to the Active Flow Table for reduction/aggregation. The `num_threads` parameter is used for implicit barrier purpose at the root of the routing tree, which guarantees all the **Updates** have been initiated. Since some operations have no need for reduction, the **Update** API is generalized with the operation code `op` for ISA translation.

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

Figure 3.2: Pseudocode of Thread Worker for Parallel PageRank.

Figure 3.2 shows the thread worker pseudocode of *pagerank* calculation loop before and after optimization, respectively. In the baseline implementation without optimization, the **atomic** update for `diff` needs to fetch the `pagerank` and `next_pagerank` value for each vertex of the graph, which consumes large amount of bandwidth due to irregular graph accessing patterns. It also needs to reduce `diff` value atomically from each thread, which causes high overhead and limit the scaling as the number of threads scales. In contrast, *Active-Routing* optimization allows updates of `diff` near data-resident locations in memory, which not only saves bandwidth but also enables bandwidth proportional MLP. In addition, the **Gather** commands from all the threads of same flow will be synchronized at a single point, the root of *ARTree*, as an implicit barrier. Then reduction is initiated by these commands and is accelerated along the *ARTree*.

3.1.2 Message Interface for Offloading

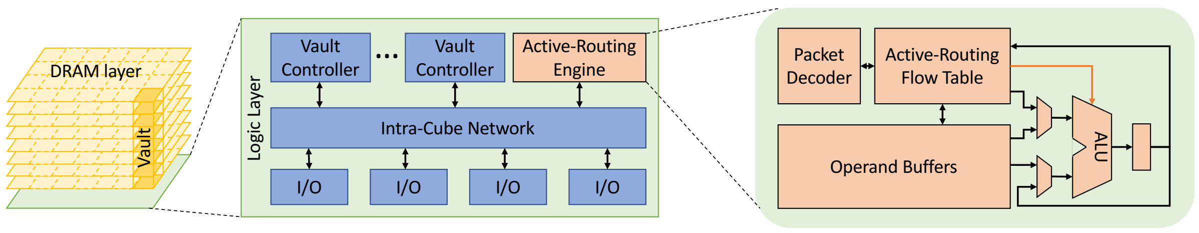
To convert the programming interface into offloading commands, the APIs are translated into extended instructions that can communicate with Message Interface (MI) shown in Figure 3.1. The **Update** and **Gather** instructions write the operand information to special registers in MI for network processing message generation. MI packetizes the command opcode and register values sent by the instruction into an **Update/Gather** packet. Then it offloads the packet into memory network for *Active-Routing* network processing. The MI functionalities can be added to Network Interface by connecting the core and router switch with marginal change without modifying the core architecture.

3.2 Active-Routing Microarchitecture

The *Active-Routing* microarchitecture is implemented in the HMC logic layer as shown in Figure 3.3 and described below.

3.2.1 Active-Routing Engine

Figure 3.3 (a) shows the processing engine that implements the *Active-Routing* functionalities, called *Active-Routing* Engine (ARE). On the HMC logic layer, Ser/Deser link I/Os and vault controllers communicate through the Intra-Cube Network, which is assumed as a switch in this work [7]. Since we adopt a unified memory network architecture [9] aiming for high network throughput, the switch is also used for forwarding packets that are not destined to the attached cube. On top of it, one extra connection is added from the switch to ARE for communication after processing. ARE can receive **Update** and **Gather** command packets as well as operand re-



(a) HMC Logic Layer for Active-Routing Engine.

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

(b) Flow Table Entry.

flowID	op_value1	op_ready1	op_value2	op_ready2
64-bit	64-bit	1-bit	64-bit	1-bit

(c) Operand Buffer Entry.

Figure 3.3: Active-Routing Microarchitecture: (a) Engine Implementation in HMC Logic Layer with (b) Flow Table Entry and (c) Operand Buffer Entry.

sponses for data processing. It also sends out responses for **Gather** commands to join its parent for *Active-Routing* reduction when the subtree originated from this cube finishes the corresponding flow.

ARE consists of a Packet Decoder, *Active-Routing* Flow Table, a pool of Operand Buffers and an Arithmetic Logic Unit (ALU). The Packet Decoder decodes the *active* command packets and schedule operations for data processing. It registers an entry in the Flow Table for new incoming *active flow* and keeps track of the packets belonging to the existing flows. It can also generate data requests for operands to the local DRAM through vault controller or remote memory cubes.

3.2.2 Active Flow Table

During *Active-Routing* processing, multiple *active flows* can co-exist in the memory network simultaneously. For example, if each thread is working on a vertex of a graph application, the number of concurrent flows will be the number of threads. Therefore, *Active-Routing* should support multiple active flows in the memory network and distinguish them. A unique flow ID is assigned for each flow that can be tracked by Active Flow Table shown in Figure 3.3 (a).

Figure 3.3 (b) displays a *flow entry* of the Flow Table. A flow entry needs to keep track of a unique flow ID, and information of number of pending/committed operations. Note that each flow will generate an *ARTree*, which is also maintained by the flow entry with its parent and children info. The entry fields are summarized in Table 3.1.

Table 3.1: Flow Table Entry Fields.

Field Name	**Purpose**
flow ID	A unique ID of the *Active Routing* flow
opcode	The operation type of this flow
result	The reduction result processed in this cube
req_counter	Count of **Update** requests for this node
rep_counter	Count of processed requests
parent	the port id connected to parent of *Active-Routing tree*
children_flags	Indicator of children of 4 cube ports
Gflag	**Gather** ready flag for *Active-Routing* reduction

3.2.3 Operand Buffers Management

For an **Update** command packet to finish processing, the offloaded operation needs to request for the operand data first before it starts to compute and commit the operation. Therefore, operand buffers are used to maintain and track the on-going **Update** operations. In general, each **Update** needs one operand buffer, especially for the operations that require two operands (e.g. `sum += A[i]`$\times$`B[i]`), since the two operand responses may arrive at the ARE from vault controllers at different times. However, for simple reduction operations such as `sum += A[i]`, there is no need for an operand buffer. The single operand response for the operation can bypass the operand buffer and be supplied to finish the reduction operation in ALU and update the result in the flow entry. This optimization frees the operand buffer resources to accelerate the two-operand operation flows.

Figure 3.3 (c) shows an Operand Buffer Entry, which keeps the flow ID to indicate which flow the **Update** belongs to and the corresponding flow entry the operation should update. It also keeps two fields to store the operand values and two ready flag bits for the operands. When a new operand response arrives to update its buffer entry, it marks the operand ready flag, and fires the operation to commit the **Update** request.

3.3 Active-Routing Processing

3.3.1 Three-Phase Processing

In general, *Active-Routing* Processing is composed of three phases as it progresses in the timeline. While sending **Update** command packets, it dynamically builds *Active-Routing tree* on-the-fly, and also starts NDP phase in parallel, followed by the **Gather** command packets. When NDP phase finishes, the **Gather** commands initiates *Active-Routing* reduction along the routing tree to finish the whole processing. Figure 3.4 shows the processing flow for different *active* packets, which will involve in either one or two phases during packet processing.

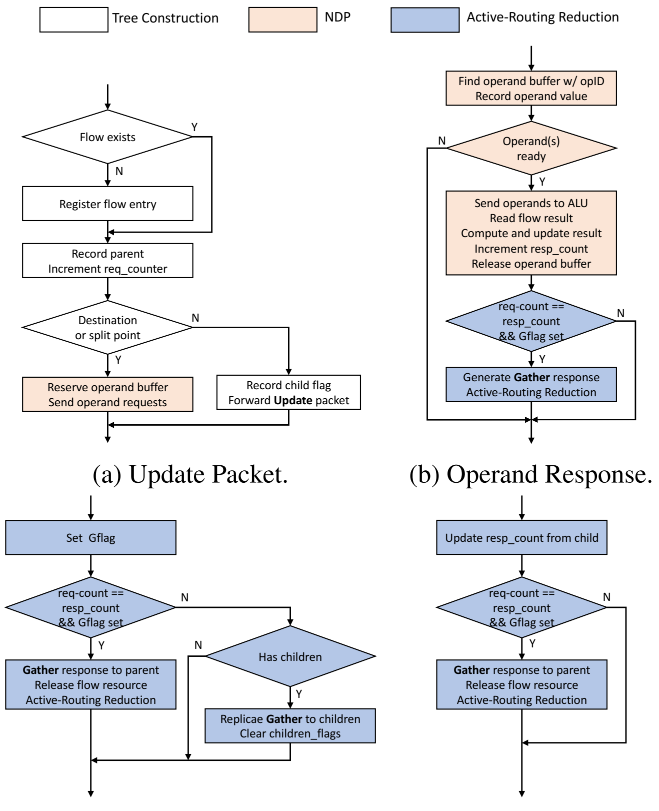


Figure 3.4: Active Packet Processing Flow Chart for (a) Update Packet, (b) Operand Response Packet, (c) Gather Request Packet and (d) Gather Response Packet.

Active-Routing Tree Construction. The first phase of *Active-Routing* Processing is to construct *ARTree*, which is dynamically built while processing **Update** requests, as shown in Figure 3.4 (a). An **Update** packet belonging to a new flow registers a flow entry to build the tree node in the cube and record its incoming link port in the parent field. If the **Update** packet is not destined for this cube or does not need to split for two-operand operation, it forwards the packet to its child and records the child field. A packet is split if both the operands do not belong to the same cube and the split location is at the last common cube on their routing path of the two operands' locations.

Near-Data Processing (Update Phase). While constructing the *ARTree*, Update phase starts Near-Data Processing at the same time. This phase involves processing of **Update** packet, and operand request/response packet as shown in Figure 3.4 (b). While processing **Update** commands whose split point of the two operands is the current cube, the processing procedure reserves an operand buffer entry and sends out operand request(s). When it receives operand responses, it records the operand value to finish the **Update** operation and commits to the flow entry.

Active-Routing Reduction (Gather Phase). Figure 3.4 (b) - (d) shows the packet processing in Gather Phase to finish *Active-Routing* reduction. The Gather Phase has one forward pass to spread the **Gather** requests from root to leaf nodes which marks the their Gflags, and a backward pass to reduce the results from leaf nodes to root node. Once the local and children's NDP finishes[1], the node replies to its parent and deallocate its flow entry. This procedure recurses from leaf nodes to the root and the whole *Active-Routing* processing commits.

3.3.2 Walking through Example

Figure 3.5 shows an example of *Active-Routing* in the memory network, where the *ARTree* is built from cube 0 as the root during the update phase. The three processing phases that constitute *Active-Routing* are illustrated below through an example of `sum += A[i]×B[i]` over a large loop, which forms an *Active-Routing flow*, as shown in Figure 3.6.

- The first phase constructs an *ARTree* on-the-fly as shown in Figure 3.6 (a), while the processors are offloading computations to the memory network through message interfaces to

[1]A parent receives **Gather** response from all of its children to indicate the completion of their NDP.

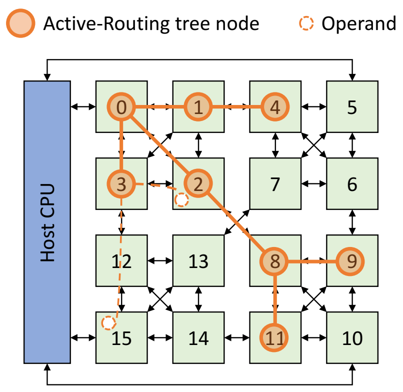
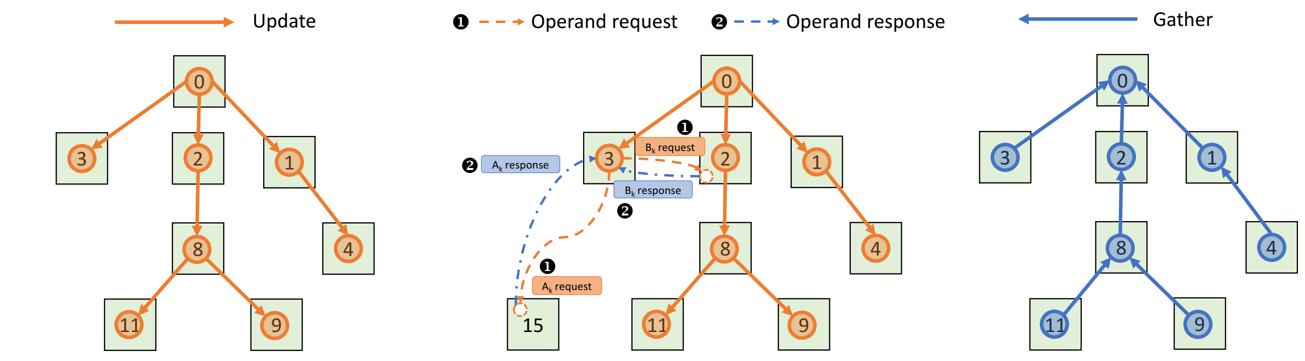


Figure 3.5: Active-Routing Example in Memory Network.



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

Figure 3.6: 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.

HMC controller. The HMC controller issues the active packets and starts the update phase at the same time.

- The offloaded computations drive Near-Data Processing during the update phase as shown in Figure 3.6 (b). Each operation `A[i]`$\times$`B[i]` needs to request its source operands `A[i]` and `B[i]` to finish the computation. Figure 3.6 (b) also shows a case where two operands do not resides in the same cube. In such scenarios, the update packet will reserve an operand buffer at the last common cube of the minimum routes (cube 3) for both operands: **1** it replicates to issue two operand requests for $\mathtt{A_k}$ and $\mathtt{B_k}$ to the resident memory cube 15 and cube 12, respectively. **2** The two operand responses are replied to cube 3 for processing and release the operand buffer.

- Figure 3.6 (c) shows the gather phase when the **Gather** packet is issued after sending all the **Update** packets. It waits for all the **Updates** in the same flow to finish and initiate network aggregation from the leaf nodes to the root along the *ARTree*.

3.4 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 with NDP offloading.

3.4.1 Virtual Memory Support

Modern processor systems use Virtual Memory and Paging to allocate memory resources to processes on demand. Each physical memory address is mapped to a different virtual address in the processes space. The applications run in the system with virtual address which is translated to physical address through TLB and page table lookup. Since *Active-Routing* is implemented by ISA extension, the offload instructions are treated as extended *active* loads/stores, so that 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. In addition, page faults can be handled as normal.

3.4.2 Cache Coherence

Multi-core systems use Cache Coherency protocols to view the most recent copies of data in each cores private caches. 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 addresses, and query for back-invalidation if data is cached on-chip similar to [2]. 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. Back-invalidation rarely happens in the simulations run in this work.

4. IMPLEMENTATION AND SIMULATIONS

Software based simulation environments are widely used in computer architecture research to generate performance projections due to infeasibility in hardware implementations. These simulators model the microarchitecture of a design under evaluation to measure the number of machine cycles required to execute program instructions. In general, a baseline configuration implemented in the same simulation environment is used as a reference for these performance projections. The level of detail incorporated by such models depends on the explicit modeling of various component modules of the architecture and may vary from simulator to simulator. One such software simulation environment with detailed organization is adopted to model the microarchitecture of *Active-Routing* to estimate the performance gains and energy efficiency achievable from this work. Section 4.1 describes the simulator choices and system modeling details of *Active-Routing* architecture implemented in this work. Later, Section 4.2 introduces benchmarks from various benchmark suites and microbenchmarks used to evaluate the architecture. The system configuration evaluated in this work is shown in Figure 3.1 and its specifications are listed in Table 4.1.

4.1 System Modeling

An execution-driven manycore simulator McSimA+ [25] with detailed microarchitecture models is used as backend for processor cores and cache hierarchy. The host CPU is configured as a 16 core out-of-order CMP with on-chip network, a two level cache hierarchy with directory based MESI coherence protocol and 4 memory slots. McSimA+ uses PIN [26], a program analysis as its front-end to extract instructions from program binaries and simulate their execution cycles in its detailed backend. PIN exposes many binary instrumentation features which are widely used to implement various tools for computer architecture analysis. A PIN tool capable of instrumenting Pthread api's is used in McSimA+ to simulate the behavior of multi-threaded applications. The tool is modified in this work to replace the Update and Gather api's in programs with equivalent x86 instruction extensions.

Table 4.1: System Configurations

Parameter		**Configuration**
CPU	Core	16 O3cores @ 2 GHz issue/commit width: 8, ROB: 64
	L1I/DCache	Private, 16KB, 4 way
	L2Cache	S-NUCA 16MB, 16 way, MESI
	NoC	4x4 mesh, 4 MC at 4 corners

Memory	DRAM Baseline	4 MCs, 64GB
		4 ranks/channel, 64 banks/rank
		tRCD=14,tRAS=34,tRP=14
		tCL=14,tBL=4,tRR=1

	HMC	4GB/cube, 4 layers 32 vaults, 8 banks/vault
	HMC-Net	16 cube Dragonfly, 4 controllers Minimal routing, virtual cut-through 16 lanes link, 12.5 Gbps/lane CrossbarSwitch clock @ 1 GHz

For HMC memory modeling, a cycle-accurate HMC simulator CasHMC [27] is integrated with McSimA+'s backend to replace its DRAM subsystem. The 4 memory controllers are updated to HMC controllers which support HMC Ser/Deser link protocol for packet based memory requests. The 16 off-chip HMCs are connected to form a pool of unified memory-network [9] with Dragonfly topology [8]. Edge links of the dragon-fly topology are connected to the HMC controllers integrated with McSimA+. Minimal routing algorithm is used to route memory requests across the memory network and virtual channels are implemented to avoid any potential network deadlocks. The microarchitectural behaviors of Active-Routing processing are implemented on the crossbar switch in HMC logic layer.

CACTI [28], a dynamic power modeling tool is used for on-chip cache power estimation. CACTI measures the power consumption of a model based on performance counters obtained from simulations. The CACTI tool is extended to incorporate memory network power which is modeled in-terms of network hops and memory accesses. This work assumes 5pJ/bit for each hop inside the memory network [29], 12 pJ/bit for HMC memory access and 39 pJ/bit access for

DRAM [30] in the baseline configuration.

4.2 Workloads

Active-Routing targets applications that have abundant reduction on data processing operations such as multiplication or pure reduction operations over a large memory footprint. To evaluate our architecture, five emerging applications from several benchmark suites are studied and four data-intensive microbenchmarks are developed for case study. In order to support execution with McSimA+ front-end, all the applications are re-implemented using Pthread library.

4.2.1 Benchmarks

Back Propagation (*backprop*) [31] is a machine learning algorithm to used train a multi-layer neural network. It has feed-forward phase and a backward weight adjustment phase. In the feed-forward pass, each node in a layer aggregates the multiplication result of the inputs from the previous layer and their corresponding connection weights for activation. For a neural network with large number of nodes, such computations bring a large amount of data movement for inputs and weights from memory to on-chip cache. Since on-chip cache is not big enough to hold all the data, it causes low data reuse and high miss rate. To solve this problem, *Active-Routing* optimization is applied for this computation phase. A neural network with a single hidden layer with 2097152 hidden units is chosen as the input model.

LU Decomposition (*lud*) [32] is a linear algebra routine that decomposes a matrix as the product of a lower and upper triangular matrices. In the kernel, it accesses the input matrix using different strides and behaves poor locality with large data set. *Active-Routing* is applied for this region of code to accelerate its execution. The input matrix dimension in the simulation is 4096.

PageRank (*pagerank*) [33] is an iterative graph analytic algorithm to compute the ranking score for each vertex in a graph and is well applied in web services. The core of the algorithm is page rank score calculation where it accumulates the average incoming scores of the pages linking to it. The irregular graph structure leads to irregular memory access patterns for such operations and can cause inefficient data reuse for big graphs. The score calculation is optimized with *Active-*

Routing and offload it to memory network. This benchmark uses web-Google graph [34] as input in this evaluations.

Matrix Multiplication (*sgemm*) [35] is a dense matrix multiplication kernel widely used in linear algebra programs. It is an important block in many library packages, such as the BLAS (Basic Linear Algebra Subprograms) and NNPACK [36], an acceleration package for neural networks on multi-core CPUs. The multiply-accumulate operations are optimized for each output elements with *Active-Routing*. To simulate large data footprint, 4096x4096 is used as the input matrix size.

Sparse Matrix-Vector Multiplication (*spmv*) [35] is a matrix vector multiplication kernel which exploits sparsity in the input matrix to reduce computation and storage. It is also an important block in BLAS. The matrix-vector multiplication loop is optimized with *Active-Routing*. The matrix dimension chosen was 4096 with 0.7 sparsity for large input size.

4.2.2 Microbenchmarks

Real world application contain segments of code which cannot be optimized by *Active-Routing*. Hence, four microbenchmarks are developed to evaluate the potential of *Active-Routing* on the pure optimization segment. Two microbenchmarks model reduction operations while the other two perform multiply-accumulate operations to model reduction over an another operator.

Reduction (*reduce, rand_reduce*) is an associative and commutative operation that is broadly used in big data applications and frameworks [37, 38]. A parallel kernel *reduce* was implemented which computes the sum of all elements of a large array sequentially. Each thread works on one partition of the array. To mimic random memory access pattern, This kernel is adapted to access the elements in the array randomly, called *rand_reduce*.

Multiply-Accumulate (*mac, rand_mac*) is widely used in machine learning community, especially deep learning applications [39, 36]. Similar to *reduce*, A multi-threaded microbenchmark *mac* was developed, which accumulates the element-wise multiplication over two large vectors. Each thread iterates from the beginning to the end of its responsible segment to model regular memory access pattern. For irregular memory accessing, they multiply two random elements of both vectors in their own segments for *rand_mac*.

5. PERFORMANCE AND POWER ANALYSIS

This work presents a novel architecture to perform reduction/aggregation operation on large-data footprints. The potential performance benefits and power efficiency that *Active-Routing* can achieve is analyzed here. Section 5.1 presents various configurations of the architecture analyzed in this work. Later Sections 5.2 and 5.3 analyze performance and power efficiency of *Active-Routing* respectively

5.1 Configurations

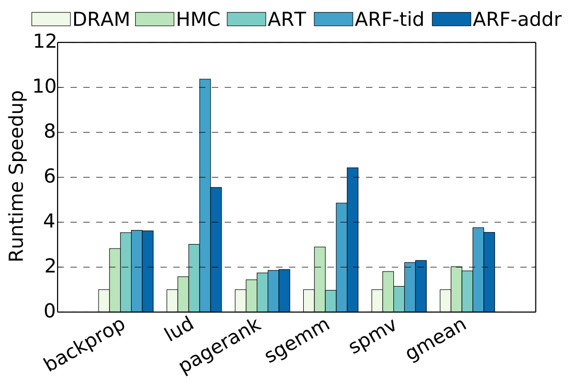
Five configuration schemes are used to understand the scope of our work in various implementations. These configurations are described as follows.

- **DRAM** system has traditional DDR memory and executes the whole program in host processor conventionally. This is considered as the baseline system for this study.
- **HMC** replaces the memory system of the baseline configuration with HMC memory network in dragonfly topology as shown in Figure 3.1. It runs the applications fully on host CPU without any NDP.
- **Active-Routing-Tree (ART)** enhances the HMC system with NDP capability to enable *Active-Routing* optimization. It constructs one tree for each flow through a static port connected to the memory network.
- **Active-Routing-Forest-tid (ARF-tid)** extends ART to construct the trees by interleaving over all the four memory network ports based on the thread ID. Multiple threads working on the same flow can generate up to four trees and work parallelly.
- **Active-Routing-Forest-addr (ARF-addr)** is an alternative of ARF-tid. Instead of building the trees based on thread ID, it constructs the trees depending on the operand addresses embedded in the **Update** packets.

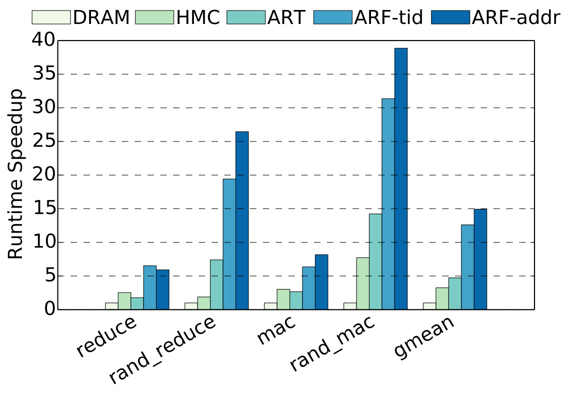
5.2 Performance

This section presents the performance analysis of *Active-Routing* over DRAM Baseline using applications and kernels from various benchmark suites and our developed microbenchmarks. First the impact on execution time is shown, then the roundtrip latency of the offloading is analyzed. Lastly, the reduction of data movement is shown.

5.2.1 Speedup



(a) Benchmarks



(b) Microbenchmarks

Figure 5.1: Runtime Speedup Over DRAM.

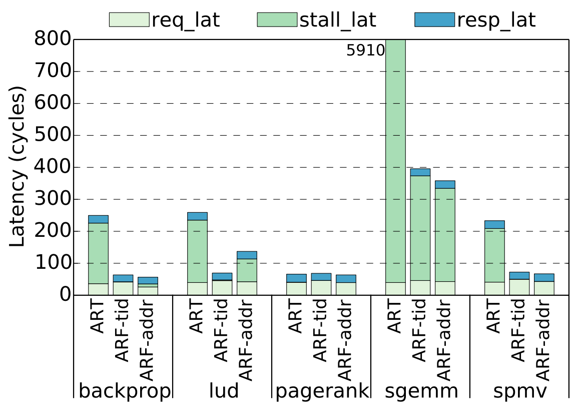
Figure 5.1 shows the execution time speedup of both benchmarks and microbenchmarks. We observe that the static ART scheme is suboptimal in some cases, even worse than HMC Baseline as in *spmv*. When it is optimized to create multiple trees from all memory access ports, called *Active-Routing Forest*, the performance improves significantly. For a neural network algorithm *backprop* and a dense matrix multiplication kernel *sgemm* and *lud*, they achieve more than 4x, 6x and 10x speedup compared to DRAM Baseline. When compared to HMC, ARF-addr improves *lud* and *sgemm* up to 6x speedup. In geomean, ARF-tid and ARF-addr help achieve more than 2x speedup for performance compared to DRAM. When compared with HMC, ARF-tid and ARF-addr improve performance by 86% and 75%, respectively. In microbenchmarks studies, the performance improvement is drastic since the whole execution is region of interest for optimization. Similarly, the static ART scheme can worsen the performance when the applications have regular memory access patterns mainly due to offloading overhead, which will be analyzed in next section. Both ARF alternatives work well across all the microbenchmarks. In irregular memory accessing scenarios, the performance can be improved even more. One interesting observation is that ARF-tid and ARF-addr may alternate the champion for different applications. Their performance depends on the balance of work distribution to the network, which will be discussed more in next section.

5.2.2 Update Offloading Roundtrip Latency

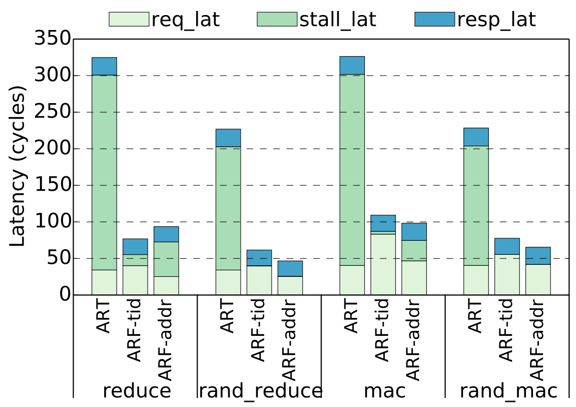
In Figure 5.2, **Update** roundtrip latency is broken into request, stall and response to help us understand the contribution of different communication components on data processing. As expected, the total latency is inversely proportional to the performance shown in Figure 5.1. It shows that static ART scheme leads to very high latency for request and stall components. This is mainly due to congestion and hotspot. In ART, all **Update** requests are sent to one memory access port which is a many-to-one communication pattern. The congestion at the port causes long stalls and propagates the pressure back leading to high request latency as well. On the other hand, ARF-tid and ARF-addr dynamically distribute the **Updates** across all available ports for tree construction. The ARF schemes can balance the load evenly and utilize the memory network resources more efficiently.

However, it is observed that the address-based ARF-addr may sometimes have higher latency due to the stalls (*lud* and *reduction*). This is because of the memory network congestion caused by load imbalance if the linear virtual memory space is not hashed well to be evenly distributed across four ports.

Figure 5.3 shows a heatmap of *lud* for ARF-tid and ARF-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. The memory access ports are placed at four corners of the memory network. In *lud*, data and tasks are distributed to all the threads evenly. When applying ARF-tid, threads distribute their generated flows to the



(a) Benchmarks



(b) Microbenchmarks

Figure 5.2: Update Roundtrip Latency Breakdown into Request, Stall and Response Latency.

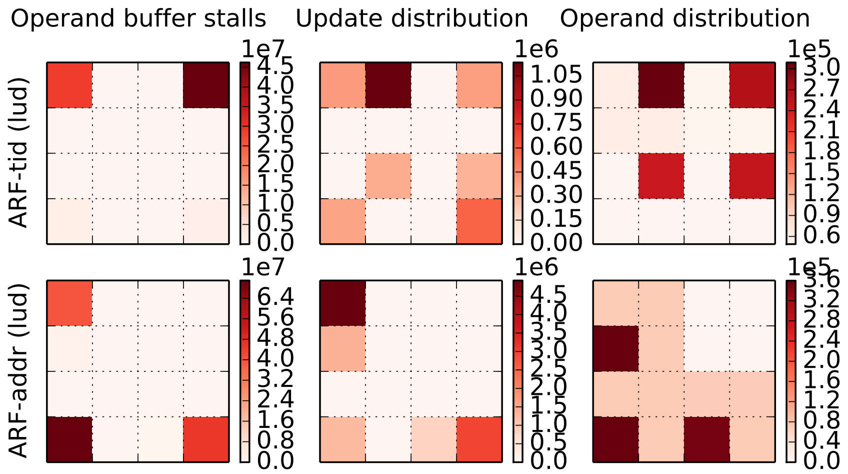


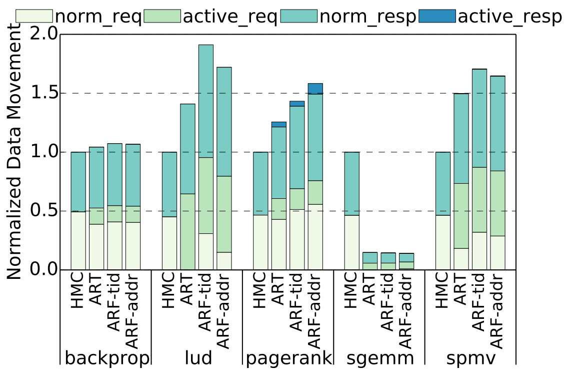
Figure 5.3: LUD Stalls and Update Distribution

memory network by interleaving the memory access ports based on their thread ID. Therefore, ARF-tid can evenly distribute the trees across the network. While in ARF-addr, active flows are distributed to the ports based on the source operands' addresses to choose the nearest port in order to reduce the hops in memory network. As shown in Figure 5.3, update distribution in ARF-tid is more balanced compared to the one in ARF-addr. Also, due to the imbalanced distribution in

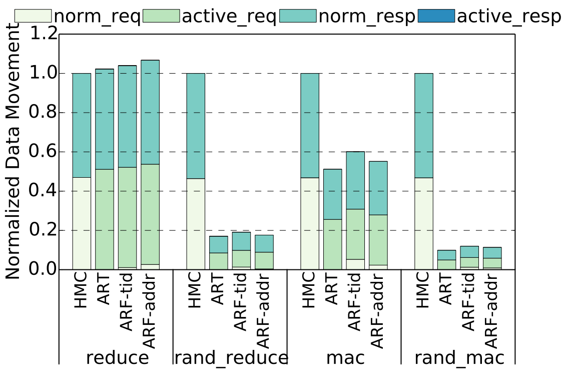
ARF-addr, it causes more stalls in the memory network, which limits performance[1].

5.2.3 Data Movement

The data movement breakdowns for normal data and active data transfer are shown in Figure 5.4. For most applications, *Active-Routing* can reduce the memory requests fetching the data, mostly source operands. However, the total data movement of *Active-Routing* schemes are higher than HMC Baseline. The region of interest for optimization is the code segment that has reduction on large amount of data processing tasks. In the benchmarks, only parts of the whole parallel phase that are evaluated are our optimization targets. The other phases still require data movement. Another overhead comes from massive **Update** fine-grained offloading.



(a) Benchmarks



(b) Microbenchmarks

Figure 5.4: On/Off-chip Data Movement Normalized to HMC.

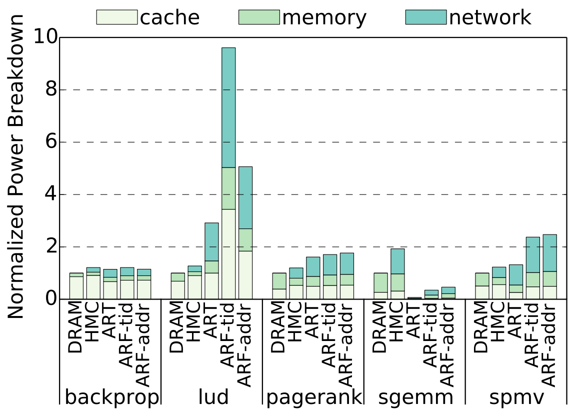
In the microbenchmarks, the whole parallel phase can be optimized and hence the data movement decreases a lot. In *reduce*, majority of its execution time is spent on summing up all the elements of a large array. Since it accesses the array elements sequentially, it exhibits very good spatial locality in its memory accesses. Good spatial locality results in high cache hits and lowers the benefit of *Active Routing*. On the other hand, in *mac*, the multiply-accumulate operation iterates over two large vectors. That may cause cache contention and conflicts due to their large

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

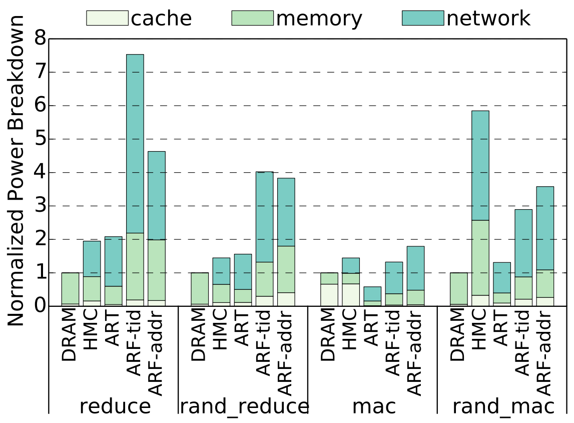
footprints. Therefore, *Active-Routing* is beneficial for cases that have abundant reductions on multiple-operand operations such as multiply-accumulate over two large memory objects. For randomly access patterns, it can reduce the waiting time of data supply which effectively reduces stalls and thus improves performance.

5.3 Power and Energy

5.3.1 Power Consumption



(a) Benchmarks

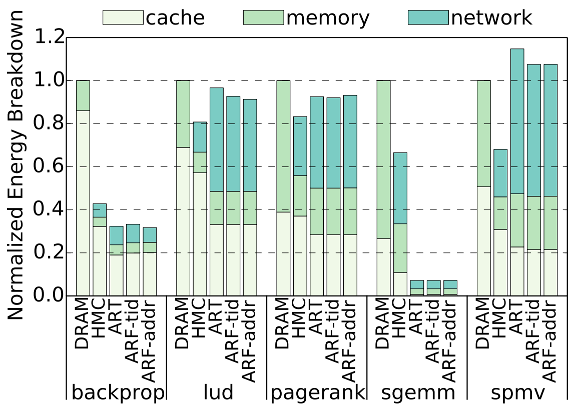


(b) Microbenchmarks

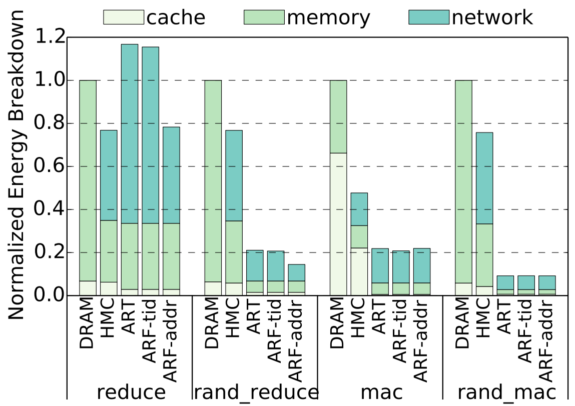
Figure 5.5: Normalized Power Consumption Over DRAM.

The power consumption breakdowns into cache, memory and memory network power are presented in Figure 5.5. It is observed that the cache power is not reduced for most of the benchmarks. This is because in the unoptimized code, the cache accessing on large data footprint can cause stall of processors thus reduce the cache access density. Whereas with *Active-Routing* optimization, the cores can issue UPDATE packets aggressively with less stalls. This increases the cache access density from the operand address loading and calculation for **Update** packets. Memory access and network power also increases due to massive processing in the memory network. Since ARF-tid and ARF-addr distribute **Updates** evenly to the network, the processing density is even higher for both benchmarks and microbenchmarks, which leads to more power consumption.

5.3.2 Energy Consumption



(a) Benchmarks



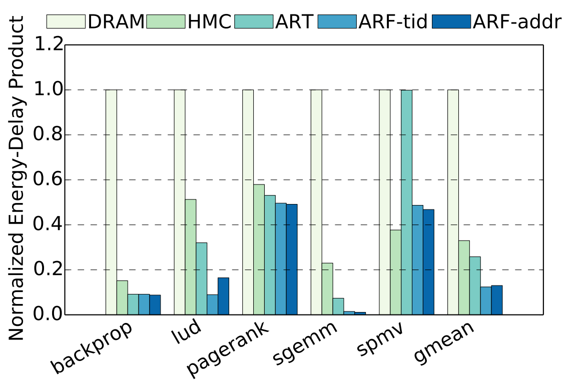
(b) Microbenchmarks

Figure 5.6: Normalized Energy Consumption Over DRAM.

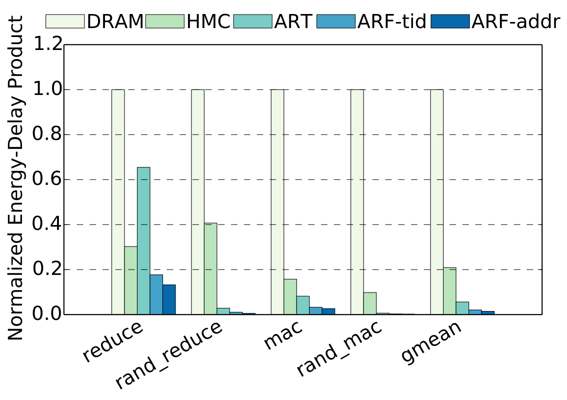
Figure 5.6 shows the energy consumptions. *Active-Routing* reduces the energy consumption in most cases, especially for microbenchmarks. For the cases that energy increases, the main contributor is the network energy. When two source operands of a data processing operation are far away in two memory cubes, they need to travel multiple hops in the network, consuming more network energy. In addition, *Active-Routing* significantly reduces cache access energy by offloading computation to memory.

5.3.3 Energy-Delay Product

Energy-Delay Product (EDP) is shown in Figure 5.7 to show the energy efficiency. It is observed that *Active-Routing* has lower EDP for all applications except for *spmv*. In *spmv*, the sparse structure can lead to input vector access with irregular stride, which highly spreads the operands across the network. This causes more network energy consumption that offsets the benefit of performance improvement. For others, even with higher energy consumption, *Active-Routing* can achieve high efficiency owing to runtime reduction from the massive parallelism in memory network. For the pure optimization region of interest in microbenchmarks, it significantly improves



(a) Benchmarks



(b) Microbenchmarks

Figure 5.7: Normalized Energy-Delay Product Over DRAM

the efficiency coming from benefits of both energy saving and execution acceleration. The energy-aware scheduling can improve the efficiency even more, which is left for future work. To summarize, ARF-tid and ARF-addr reduce the EDP for benchmarks by 75% and 88% on average compared to HMC Baseline.

5.4 Dynamic Offloading: A Case Study

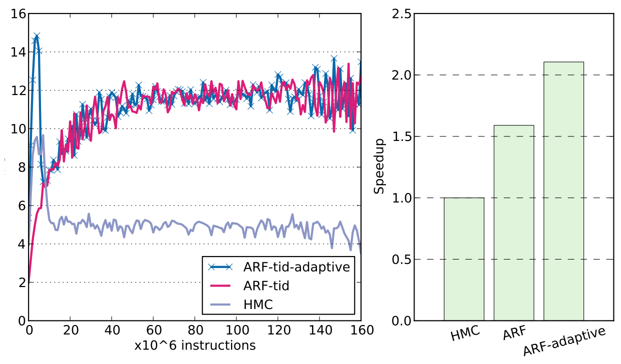


Figure 5.8: LUD Phase Analysis and Dynamic Offloading

This section shows with the help of an example that the performance can be further improved using application information. *Active-Routing* can be enhanced with a runtime knob, which decides whether to offload **Updates** dynamically on the basis of memory access and communication patterns to achieve more performance gains. Execution phases that exhibits good locality experience performance benefits by exploiting cache hits when scheduled on the host processor. In *lud*, the working set in each iteration keeps increasing during its kernel execution. At the beginning, the application benefits from locality. And as the iterations increases, it walks through larger data footprint and breaks the locality. This can be captured from the relationship between **Updates** per flow and cache parameters. For such program behavior, the best execution model will be processing in the host at the beginning and adapt to *Active-Routing* processing as it progresses and exhibits low reuse memory access pattern. This case study analyzes *lud*'s phase behaviors as shown in Figure 5.8. At the beginning phases, the IPC of HMC is higher than ARF-tid. When it proceeds to a later phase, ARF-tid catches up and outperform HMC. A mix of two, named ARF-tid-adaptive is run, which executes in host processor before the crosspoint and later offload for *Active-Routing* processing. For this analysis, a matrix size of 256 was used as input data size and *Active-Routing* offloading is enabled when Updates per flow is higher that a threshold $(\frac{CACHE\_BLK\_SIZE}{stride1} + \frac{CACHE\_BLK\_SIZE}{stride2})$. The ARF-tid-adaptive improves performance from 1.5x to 2.0x.

6. CONCLUSIONS AND FUTURE WORK

This work proposes *Active-Routing*, an In-Network Compute Architecture enabling compute on the way for Near-Data Processing, to accelerate reduction on data processing operations in data-intensive applications. *Active-Routing* is implemented as a novel three-phase processing mechanism: *Active-Routing tree* Construction, Data Processing and *Active-Routing* Reduction. It moves the computation near data in the memory network for processing and aggregates the results along their routing paths. Our evaluation results show that *Active-Routing* can achieve up to 6x speedup with a geometric mean of 75% performance improvement and reduce energy-delay product by 88% on average, compared to a system integrated with a die-stacked memory network.

To extend this work, various granularities for *Active-Routing* offloading and energy-aware scheduling to amortize the data movement and energy overhead can be explored. In addition, a model can be formalized to categorize memory access patterns and communication patterns and use it to design a runtime-aware architecture for dynamic offloading to push performance improvement further.

REFERENCES

[1] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[2] 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.

[3] 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.

[4] 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.

[5] P. Kogge, "A Short History of PIM at Notre Dame." Available from https:://www.cse.nd.edu/∼pim/projects.html, 1999.

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

[7] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance," in *Symposium on VLSI Technology (VLSIT)*, pp. 87–88, 2012.

[8] 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.

[9] 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.

[10] 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.

[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] A. Jaleel, M. Mattina, and B. Jacob, “Last Level Cache (LLC) Performance of Data Mining Workloads on a CMP - A Case Study of Parallel Bioinformatics Workloads,” in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 88–98, Feb 2006.

[13] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, “BigDataBench: A Big Data Benchmark Suite from Internet Services,” in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 488–499, Feb 2014.

[14] 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.

[15] 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.

[16] 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 Opera-

tions Using Commodity DRAM Technology," in *International Symposium on Microarchitecture (MICRO)*, pp. 273–287, ACM, 2017.

[17] G. Zhang, W. Horn, and D. Sanchez, "Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-Coherent Systems," in *International Symposium on Microarchitecture (MICRO)*, pp. 13–25, ACM, 2015.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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 Inter-

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization (IISWC)*, pp. 44–54, IEEE Computer Society, 2009.

[32] 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.

[33] 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.

[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] M. Dukhan, "NNPACK." Available from https:://github.com/Maratyszcza/NNPACK, 2017.

[37] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Symposium on Opearting Systems Design & Implementation (OSDI)*, pp. 10–10, USENIX Association, 2004.

[38] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, IEEE, 2010.

[39] 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.