Title of Dissertation: Design Space Exploration of Data-centric Architectures

Name of Candidate: Smriti Prathapan
Ph.D. in Computer Science, 2020

Dissertation and Abstract Approved: 
Dr. Milton Halem
Research Professor
Department of Computer Science and Electrical Engineering

Date Approved: 

APPROVAL SHEET
ABSTRACT

Title of dissertation: Design Space Exploration of Data-centric Architectures

Smriti Prathapan, Doctor of Philosophy, 2020

Dissertation directed by: Dr. Milton Halem
Research Professor
Department of Computer Science and
Electrical Engineering, UMBC

The era of "big data" is leading to changes in the compute paradigm, in particular to the notion of moving computation to data, known as Near Data Processing (NDP). Technological advancements have enabled the application of NDP at many levels of the memory hierarchy from cache to DRAM, from non-volatile storage-class memory to processors embedded in storage devices. This dissertation explores the effectiveness of data-centric compute architectures using Active Storage, Processing-in-Memory and Coherent Access Processor Interface (CAPI) accelerated Flash storage. We developed a compute framework Active In-Storage (AiSTOR) that enables scalable distributed Big Data Processing by directly performing the computations on active storage devices. AiSTOR has the following three major advantages: (i) active storage utilizes the processor capabilities on the storage devices and this significantly reducing the bandwidth requirement of the network, (ii) the computations can take advantage of the inherent map/reduce parallelism by using the array of the distributed storage processors available on the active data storage devices, thereby aggregating the processing power of a cluster of active devices, (iii) it can perform coherent processing of streaming data as it arrives on the storage devices. We define a generic NDP architecture which is well-suited for memory-bound computations and implement the software kernels for NDP-based algorithmic mapping. We show for a modest sized NDP system, that the AiSTOR architecture framework employing distributed processing algorithms can yield efficient and accurate computational processing performance. In comparison with Hadoop based MapReduce, the compute times on AiSTOR has significant performance benefits by up to 18%, while providing very competitive results compared to Spark-based in-memory processing. The effectiveness of the NDP architecture is demonstrated
by evaluating the row-buffer management policies (open-page and closed-page) with the controller modifications and a generic unmodified architecture. The PIM open-page policy has 52% higher operation throughput than the host and 3.7% higher throughput and 50% lesser DRAM activations than PIM-closed page policy. Further, this dissertation explores the potential impact of hidden CPU usage in handling the IO requests in heterogeneous storage systems when using CAPIFlash library and finding the optimal IO/s and OP/s for heterogeneous storage memory devices such as NVM, SSD and RAM. FS900 with CAPI, when using the RAM metadata cache, performed 2x as many read OP/s in synchronous mode and 3x in asynchronous mode. SSD and NVM had 66% higher IO/s in comparison with RAM in asynchronous mode.
Design Space Exploration of Data-centric Architectures

by
Smriti Prathapan

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland Baltimore County, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2020
To my dad, Prathapan Neelakantan - for being a dreamer. You are my hero!

To my dear husband, Dileep, for all your patience and love.
Acknowledgments

I would like to express my immense gratitude to Dr. Milton Halem, my advisor, for believing in my abilities and giving me an opportunity at the Center for Hybrid Multicore Productivity Research (CHMPR). I thank him for his encouragement and support throughout this work. He was always able to find financial assistance to sponsor me. I am grateful for his valuable guidance, resources and financial support.

I am profoundly thankful to Dr. John Dorband for his valuable advice, help and support. I thank him for always helping me overcome technical difficulties and provide access to compute clusters, allocating storage and resolving compute cluster related problems.

I am very thankful to Dr. Navid Golpayegani for being an external member of my committee and for his collaboration, guidance and support throughout this work.

I thank Dr. Yelena Yesha and Dr. Charles Nicholas for willing to serve on my committee as well as for their valuable advice, help and support.

I would like to thank Bryan Wyatt, Robert Warmka, Jon Trantham, and Chris Markey from Seagate for providing me with hardware and software support and CHMPR membership support.

I would also like to acknowledge lecturers, colleagues, and all of my lab-mates from Center for Hybrid Multicore Productivity Research (now CARTA) especially, Kaushik Velusamy, Shruti Rao, Yin Huang and Dorsa Ziaei.

Thank you to my family, particularly my parents and my mother-in-law for their encouragement. Most importantly, to my husband Dileep and daughter Eva, for traveling with me through the challenges and being extremely patient through the many struggles and tolerating the long work hours. Thanks to Eva for waking up early in the mornings to drive with me to UMBC.
# TABLE OF CONTENTS

**LIST OF FIGURES** ................................................................. ix

**Chapter 1**  INTRODUCTION .................................................... 1

1.1 Motivation ................................................................. 4
1.2 Thesis Statement .......................................................... 6
1.3 Contributions ............................................................. 6
1.4 Thesis Organization ....................................................... 8

**Chapter 2**  RELATED WORKS .................................................. 9

2.1 In-Storage Processing (ISP) .............................................. 9
2.1.1 Database Machines .................................................... 9
2.1.2 Intelligent Disks (IDISKs) ........................................... 10
2.1.3 Active Disks in the 90s .............................................. 10
2.1.4 Flash based Solid State Devices (SSDs) ......................... 11
2.2 Processing in Memory (PIM) ............................................. 13
2.2.1 Intelligent RAM (IRAM) ............................................ 13
2.2.2 NDP for In-memory Analytics Frameworks ..................... 14
2.2.3 Active Storage based on HDFS ................................... 14
2.2.4 AnalyzeThis: Work-flow aware Storage System .............. 14
3.7 Conclusion ................................................................. 38

Chapter 4 PROCESSING IN MEMORY ARCHITECTURE FOR MEMORY-BOUND COMPUTATIONS ........................................... 41

4.1 Introduction ............................................................... 41

4.2 Limitations of current Architectures ................................. 43

4.2.1 Memory Wall ......................................................... 43

4.2.2 Bandwidth Wall ..................................................... 43

4.2.3 Data Movement .................................................... 44

4.2.4 Power Wall ......................................................... 45

4.3 Memory Technology Background .................................... 46

4.3.1 DRAM ............................................................... 46

4.3.2 3D-stacked Memories ............................................. 47

4.3.3 High Bandwidth Memory ......................................... 48

4.3.4 Hybrid Memory Cube ............................................. 48

4.4 Related Work ............................................................. 49

4.5 NDP Architecture ....................................................... 52

4.5.1 PIM Software Stack ............................................... 53

4.5.2 Program on PIM .................................................... 54

4.5.3 PIM controller Design ............................................ 55

4.6 Evaluation ................................................................. 56

4.6.1 Accelerating KNN on a Smart Memory Cube ................. 56

4.6.2 KNN Characterization ............................................ 58

4.6.3 Mapping KNN on a PIM system ................................ 60

4.6.4 Performance ......................................................... 62
4.7 Summary ................................................................. 66

Chapter 5  COHERENT ACCELERATOR PROCESSOR INTERFACE FOR HETEROGENEOUS STORAGE AND MEMORY DEVICES 68

5.1 Introduction .......................................................... 68
5.2 Background .......................................................... 69
5.3 CAPIFlash Design .................................................... 71
  5.3.1 CAPI Adapter ....................................................... 72
  5.3.2 IBM FlashSystem 900 .............................................. 72
  5.3.3 NVM and SSD ..................................................... 73
  5.3.4 RAM ............................................................... 74
  5.3.5 Processor ........................................................ 74
5.4 Evaluation ............................................................ 74
  5.4.1 Dataset .......................................................... 74
  5.4.2 Performance Metrics .............................................. 75
  5.4.3 CAPI Flash parameters ......................................... 75
  5.4.4 Benchmark Application ........................................... 77
  5.4.5 Performance of Write Operations ............................... 77
  5.4.6 Performance of Read Operations ............................... 78
  5.4.7 Summary of Read Write performance of RAM, FS900, NVM and SSD .... 81
  5.4.8 Performance of \texttt{ark\_nextN} API ........................... 85
  5.4.9 Large Experiments Results ...................................... 86
5.5 Related Works ....................................................... 87
5.6 Conclusions ........................................................ 87
Chapter 6  CONCLUSION AND FUTURE WORK  

6.1 Summary of Contributions  

6.2 Future Work  

REFERENCES
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Storage stack and Compute Cluster with Active Storage</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>AiSTOR - a data-centric compute architecture</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>MapReduce Execution flow on ISP Devices</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>Performance Summary of applications on ADs cluster</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Performance Summary of MapReduce on Hadoop, Spark and AD cluster</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>DRAM and I/O subsystem</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>NDP architecture with 3D-stacked memory</td>
<td>51</td>
</tr>
<tr>
<td>4.3</td>
<td>PIM Software Stack</td>
<td>54</td>
</tr>
<tr>
<td>4.4</td>
<td>PIM Controller Design</td>
<td>55</td>
</tr>
<tr>
<td>4.5</td>
<td>Mapping KNN algorithm on a PIM system</td>
<td>60</td>
</tr>
<tr>
<td>4.6</td>
<td>KNN Execution times on host and PIM</td>
<td>63</td>
</tr>
<tr>
<td>4.7</td>
<td>PIM Operation Throughput</td>
<td>64</td>
</tr>
<tr>
<td>4.8</td>
<td>PIM Memory Accesses</td>
<td>65</td>
</tr>
<tr>
<td>5.1</td>
<td>Write performance for FS900, RAM, NVM and SSD</td>
<td>79</td>
</tr>
<tr>
<td>5.2</td>
<td>Read performance for FS900, RAM, NVM and SSD</td>
<td>80</td>
</tr>
<tr>
<td>5.3</td>
<td>Peak performance of FS900, RAM, NVM and SSD for 10 Million KV dataset</td>
<td>82</td>
</tr>
</tbody>
</table>
5.4 IO/s performance of FS900, RAM, NVM and SSD for 30 Million KV dataset ... 83

5.5 Read Time and Write Time of FS900, RAM, NVM and SSD for 30 Million KV dataset 83

5.6 CPU% used by the application process for 30 million KV dataset .................. 84

5.7 CPU% used by the system process for 30 million KV dataset ..................... 84
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>Advanced RISC Machine</td>
</tr>
<tr>
<td>AD</td>
<td>Active Drive&lt;sup&gt;TM&lt;/sup&gt; Devices</td>
</tr>
<tr>
<td>ADFS</td>
<td>Active Disk File System</td>
</tr>
<tr>
<td>AFAn</td>
<td>All-Flash Array.next</td>
</tr>
<tr>
<td>AFEs</td>
<td>Active Flash Elements</td>
</tr>
<tr>
<td>AFU</td>
<td>Accelerator Function Unit</td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>AiSTOR</td>
<td>Active In-Storage</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>AT</td>
<td>Assistive Technologies</td>
</tr>
<tr>
<td>CAPI</td>
<td>Coherent Accelerator Processor Interface</td>
</tr>
<tr>
<td>CGRA</td>
<td>Coarse-Grain Reconfigurable Arrays</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSD</td>
<td>Computational Storage Devices</td>
</tr>
<tr>
<td>CXL</td>
<td>Coherent Accelerator Interface</td>
</tr>
<tr>
<td>DDR</td>
<td>Double Data Rate</td>
</tr>
</tbody>
</table>
**DIMMs**  Dual In-line Memory Modules

**DMA**  Direct Memory Access

**DRAM**  Dynamic Random Access Memory

**ESS**  Elastic Storage System

**FPGA**  Filed Programmable Gate Array

**GPU**  Graphics Processing Unit

**HBM**  High Bandwidth Memory

**HDD**  Hard Disk Drive

**HDF**  Hierarchical Data Format

**HDFS**  Hadoop Distributed File System

**HMC**  Hybrid Memory Cube

**HPC**  High Performance Computing

**IDISK**  Intelligent Disks

**IRAM**  Intelligent RAM

**ISP**  In-Storage Processing

**KNN**  K-Nearest Neighbor

**KV**  Key Value

**LLC**  Last Level Cache

**MMU**  Memory Management Unit
MODIS  MODeate-resolution Imaging Spectrometer

NAS    Network Attached Storage

NDP   Near Data Processing

NMC  Near-Memory Computing

NVM  Non-Volatile Memory

NVMe Non-Volatile Memory Express

OSD  Object-based Storage Devices

PCBA  Printed Circuit Board Assembly

PCIe Peripheral Component Interconnect Express

PFS  Parallel File System

PHB  PCIe Host Bridge

PIM  Processing In Memory

POSIX Portable Operating System Interface

PSL  Power Service Layer

QD  Queue Depth

QLC  Quad-Level Cell

RAID Redundant Array of Inexpensive Disks

RDD Resilient DistributedDatasets

RTL  Register Transfer Level
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAN</td>
<td>Storage Area Network</td>
</tr>
<tr>
<td>SAS</td>
<td>Serial Attached SCSI</td>
</tr>
<tr>
<td>SATA</td>
<td>Serial Advanced Technology Attachment</td>
</tr>
<tr>
<td>SDS</td>
<td>Software-Defined Storage</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SMC</td>
<td>Smart Memory Cube</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multi-Processor</td>
</tr>
<tr>
<td>SoC</td>
<td>System on a Chip</td>
</tr>
<tr>
<td>SPM</td>
<td>Scratchpad Memory</td>
</tr>
<tr>
<td>SSD</td>
<td>Solid State Drive</td>
</tr>
<tr>
<td>TLB</td>
<td>Translation Look-aside Buffer</td>
</tr>
<tr>
<td>TSV</td>
<td>Through-Silicon Via</td>
</tr>
<tr>
<td>WWPN</td>
<td>World Wide Port Name</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

In the era of Big Data, 2.5 quintillion ($10^{18}$) bytes of data are created everyday (Reinsel, Gantz, & Rydning 2020). Humans and machines contribute the increase in data. In particular, applications in the areas of meteorology, genomics, connectomics, physics simulations, biology, information sensing Internet of Things devices and wireless sensors are contributors to the exponential growth of data. In past years, the volume and speed of data creation have consistently been increasing. Estimates indicate that 90% of all the data in the world was produced within the past two years (Marr 2018). A fundamental challenge for Big Data Applications is to scan large volumes of data and extract useful information or knowledge (Rajaraman & Ullman 2011) (Wu et al. 2014) (Oussous et al. 2018). As applications become more data intensive, compute systems supporting Big Data are required to have larger memory footprints. A typical algorithm scanning Big Data usually requires all the data to be loaded to the main memory. This is clearly a technical barrier, since moving data across different locations is expensive owing to network communication and IO costs. Thus, a Big Data processing framework, where data needs to be transferred between processor, memory subsystem and storage clusters, will have a severe impact on I/O performance and energy efficiency.

Data movement between the storage and compute nodes have in recent years become a critical factor affecting performance and is an active area of research. As storage systems become larger and more complex, transferring data between remote compute and storage tiers becomes impractical.
Clusters and cloud storage applications that perform computation on big data typically separate the storage from the compute clusters, as the requirements of the compute and storage tiers are different from each other. A serious drawback for any such architecture is the need to move large amounts of data from the storage nodes to the compute nodes in order to perform computations. Such movements are frequently followed by moving even larger data set results back to the storage cluster. In large scale data analytics systems, it is critical to minimize data movement not only to avoid overall performance degradation owing to latency, but also to enhance power-efficiency and reliability. With the data archival expected to reach 150 Pb/year between 2021-2023 and projected to increase to 400 Pb/year by 2025 (Davis.Michael.2019), moving large chunks of data between storage and compute nodes is highly inefficient in the Exabyte storage era.

Another aspect of data movement is the frequent data transfers between processor units and memory, which cause high energy consumption and has become a growing concern for data-intensive applications. The discrepancy between processor speeds and memory access speeds (memory wall (Wulf & McKee 1995)) results in memory access to be the primary performance bottleneck in most of today’s data-intensive applications. Most compute architectures rely on multi-level caches to reduce the data access latency. Limitations of on-chip cache memories and main memory bandwidth of the conventional processing units contribute to the inefficiencies in processing big data. As the applications use datasets which exceed the size of RAM and when data access is non-uniform and sparse (Velusamy, Prathapan, & Halem 2019), the existing technologies fail to reduce the frequent data movement between memory and processor units.

The traditional compute-centric model for large-scale data processing is shifting to more of a data-centric model. Handling large data volume to the locations where computations are performed drives up the cost of data processing. In the compute-centric model, data resides on the disk, and moves as needed to a central computing cluster across a deep storage hierarchy. This model is sufficient as long as computation dominates data movement. In a data-centric model, the data lives
in different storage and memory units within the hierarchy, with processing engines surrounding the
data and operating directly on such data without moving data across the system.

Data movement is the major factor impacting latency, throughput performance, power-efficiency,
and reliability of a system. The best approach to reduce the overhead of data movement is to
minimize or even avoid it altogether, when possible. Thus, there is a need to explore approaches
that focus on moving computations closer to data, often referred to as Near Data Processing (NDP)
(Balasubramonian et al. 2014).

One approach of NDP is to equip the memory or storage with local intelligence (i.e. processors
and network channels) thereby allowing computations on the data. Alternative approaches include
sending small compute kernels from the processor to intelligent data nodes. Another example of a
highly successful approach to NDP has been the development of Apache Spark that moves data to
the processor’s RAM prior to use. NDP seeks to minimize data transfers and power consumption
and by computing at the most appropriate location in the storage hierarchy. Computations can be
performed in memory or in the storage device and memory units where the input data resides (Chi et
al. 2016). The different realizations of NDP includes processing in memory (PIM) (Gokhale, Holmes,
& Iobst 1995) and in-storage processing (ISP) (Acharya, Uysal, & Saltz 1998; Kim et al. 2016;
Lee et al. 2016b; Choi & Kee 2015). ISP, also known as active storage (Riedel et al. 2001), aims
to expose computational elements within the storage infrastructure to perform general-purpose
computation on data.

The conventional homogeneous multi-core processors are unable to provide the continued
performance and energy improvement to support the emerging of massive and dynamic datasets
(Jia et al. 2017). Heterogeneous computing architectures that feature hardware accelerators are
emerging as a promising paradigm to better exploit and optimize the IO sub-system. Among different
heterogeneous device are the High Bandwidth Memory (HBM)-enabled FPGA accelerators (Jun et
al. 2017) address these challenges to provide a large memory bandwidth. In contrast to traditional
I/O-based co-processors that require explicit data movement, coherently attached accelerators such as FPGA attached via a Coherent Accelerator Processor Interface (CAPI) to an IBM POWER8 processor (Stuecheli et al. 2015a; 2018) can operate on the same virtual address space than the host CPU. Non-Volatile Memory technologies such as Intel Optane® DC™ features high density, DRAM comparable performance, and persistence (Peng, Gokhale, & Green 2019). These characteristics position byte-addressable Non-Volatile Memory (NVM) as a promising alternative for large-scale graph processing which often have to be distributed over multiple compute nodes to enable in-memory processing (Malewicz et al. 2010).

The ever increasing demand for large amounts of memory/storage medium and steady advances in 3D technology has created Quad-Level Cell (QLC) 3D NAND with 4 bits/cell storage offering more capacity and lower cost-per bit (Takai et al. 2019; Lee et al. 2018). IBM Flashsystem (including Storwize) solid-drive disk storage system that is scalable with high storage density to keep up with growing data demands (Flashsystem ). IBM’s Elastic Storage System (ESS) which is scalable up to yottabyte (1,000 Zetabytes or $10^{24}$ bytes) provides fast analysis of large amounts of storage and is aimed at potentially supporting high performance Artificial Intelligence (AI), big data and HPC workflows (Coughlin ). Advances in storage technology to accelerate analytics, AI, and deep learning include a petabyte scale, high performance All-Flash Array.next (AFAn) system which blends Intel® Optane™ performance with the capacity and cost-effectiveness of QLC flash drives (All.Flash.Intel.Optane ).

1.1 Motivation

As the novel big data workloads of cognitive computing and machine learning require embarrassingly parallel and highly efficient processor architectures, a shift from compute-centric to data-centric system is inevitable. With the approaching end of Moore’s law, innovative novel architectures are required to tackle exascale and beyond since the computing systems face the power, memory, and
bandwidth walls. Many data-intensive applications in the areas of material science, astronomy, social computing, bioinformatics etc., need newer architectures. Several NDP architectures have been proposed to enhance the performance of applications such as graph processing, pointer chasing, Database scan and filtering.

Building on the shift towards data-centric compute systems, the concept of Near Data Processing (NDP) seems to be the most promising. Co-locating compute tasks on bandwidth-rich in-memory and in-storage processing units mitigates data movement by avoiding the traversal through the entire memory/storage hierarchy thereby, improving the power efficiency and compute performance. The evolution of software and hardware led system designers to create new compute architectures to tackle the problems of data movement. While ISP reduces the data movement between storage and compute units by using compute-capable storage devices, PIM couples compute units closer to the data by using the 3D die-stacking technology allowing the logic circuits, such as simple processors, to be placed in close proximity to memory, using high-bandwidth interconnects.

ISP uses Active or Computational Storage Devices (CSDs) which consist of an embedded processor on the storage device. This is designed to perform computations directly on the data within the storage device. Processing the data within the storage node reduces the data transfer from the storage server to the client and applications can take advantage of the parallelism in large storage systems. The limitations of file-based systems and data management in a traditional compute-centric environment can be overcome to an extent by using an active storage enabled with Open Ethernet Drive approach. Since the processing can be done directly on the storage device taking advantage of the scalability offered by object storage, data intensive applications benefit from a significant reduction in I/O by avoiding data transfer over the network. This approach explores the inherent data parallelism of the data in the storage devices by utilizing the available processor cluster power.

In contrast, Processing-In Memory (PIM) or Near-Memory Computing (NMC) aims at processing close to where the data resides by compute elements close to the data in the memory units.
The 3D-stacking enables processing close to the memory. It allows the stacking together the logic and memory using Through-Silicon Via’s (TSVs) thereby, reducing memory access latency, power consumption and provides a higher bandwidth (Jeddeloh & Keeth 2012).

Three approaches for near data processing are explored in this thesis: In-storage processing (ISP) using Active Storage, Processing-In-Memory (PIM) using a Smart Memory Cube simulator and Coherent Access Processor Interface (CAPI) accelerated Flash storage (Stuecheli et al. 2015a).

1.2 Thesis Statement

We develop, explore and evaluate an Active In-Storage (AiSTOR) Near Data Processing technology for mitigating Big Data computing inefficiencies by taking advantage of the computational power of in-storage processing devices, near-memory processors and coherent processing software systems.

We simulate a Processor in Memory Architecture and show how I/O-intensive and memory-bound applications can improve their overall performance and efficiently balance the use of system wide resources.

1.3 Contributions

AiSTOR: Implemented an Active In-Storage compute framework for program execution within the ISP devices

AiSTOR is a compute framework for scalable distributed big data processing using In-Storage processing architecture. AiSTOR provides transparent and multi-granularity processing for program execution within the ISP devices.

Chaining multiple functions: AiSTOR is based on a stream based programming model which provides a simple and convenient way for users to apply several application functions on the data object, a set of commonly used functions are included in an applet. This facilitates ease of programming
since the user can easily access this functionality by passing the function name and the required attributes. It also enables chaining multiple functions on a single object.

Resource Monitoring: AiSTOR performs runtime resource monitoring by loading an executable to fetch the applet status while it is running on the storage device. The runtime resource monitoring metrics such as CPU utilization on the storage device and the temperature of the processor cores could be used for load balancing.

Security Considerations: To ensure the safety of the overall system, two aspects of security are considered: bad code in the applet and isolating the data between users.

Data Allocation: In a Big Data application using a large dataset, data allocation becomes a bottleneck, i.e. to decide in which storage device to allocate a given key/value pair. For a smaller dataset, it is easier to have a metadata file to maintain a small-sized indexing table, which contains the IP address of device and the corresponding key(s). The disadvantage of maintaining a metadata file is, as the data grows larger, the metadata file gets bigger. For a terabyte-sized dataset, the metadata file could end up to be a performance bottleneck. To avoid this, we used a deterministic approach of data allocation. This scheme has the following advantages: (1) It distributes the keys uniformly across all the devices. (2) It eliminates the need of a metadata file since knowing prefix range for the keys and the device number for each device in a storage cluster, the key distribution for each device could be determined.

**PIM: Performed Near Memory Acceleration of Memory-bound algorithms** Designed a controller for a simulated Near Data Processing (NDP) architecture to support memory-bound computations and implement the software kernels for NDP-based mapping for the algorithms on a cycle accurate full-system NDP architecture framework with detailed performance analysis. Evaluate the row-buffer management policies (open-page and closed-page) on an NDP architecture with the new controller design and a generic unmodified architecture. With the proposed modifications, we obtained upto 75% performance gain for a KNN algorithm in comparison with the host. Further, this
study provides a characterization of the NDP-mapped memory-bound algorithms and demonstrates the shortcomings of other implementations.

**CAPI: Analysis of Coherent Accelerator Processor Interface for Heterogeneous memory/storage devices**

Analyze the impact of CPU usage in handling the IO requests in heterogeneous storage systems when using CAPIFlash library and finding the optimal IO/s and OP/s for heterogeneous storage devices such as NVM, SSD and RAM. For applications that require high computational power, multi-core processors often do not suffice to provide high performance. The use of hardware accelerators such as GPUs, ASICs, and FPGAs can provide increased performance for massively parallel programmable architectures for a wide range of applications. The performance of a CAPI accelerator was evaluated by varying the threads and other parameters of the CAPIFlash library and comparing it with RAM, SSD and NVM without using the CAPI accelerator. This work is the first to study the performance of the FS900 with CAPI accelerator card through CAPIFlash read write benchmark.

1.4 Thesis Organization

This thesis is organized as follows:

Chapter 2, reviews the Near Data Processing approaches for PIM and ISP as well as other frameworks which facilitate computations closer to data. In Chapter 3, we describe the design and implementation of Astor - a scalable and distributed In-Storage Processing framework, Chapter 4 explores near-memory processing and accelerating memory-bound algorithms using 3D-stacked memory. Chapter 6 discusses the behavior of other heterogeneous storage and memory devices such as NVM, SSD and RAM. Chapter 6 presents the conclusion and the future of Near Data Processing.
Chapter 2

RELATED WORKS

2.1 In-Storage Processing (ISP)

Several authors at the end of the 1990s (Keeton, Patterson, & Hellerstein 1998) (Riedel, Gibson, & Faloutsos 1998) developed the idea of intelligent storage. Similar ideas were proposed even earlier in the 80s (DeWitt & Hawthorn 1981; Hurson et al. 1989) in the context of databases. The original research efforts were based on the premise that modern storage architectures would have progressed to a point that there would exist the real possibility of utilizing the processing power provided by the drive controller itself. However, for numerous reasons, commodity disk vendors have not offered the required software support and interfaces to make intelligent storage practical and widely used.

This section details the trends in technology that has enabled processing on storage medium such as Hard Disk Drives (HDDs) and Solid State Drives (SSDs). Further, it details the trends in user and workload characteristics of data-intensive applications.

2.1.1 Database Machines

The idea of performing computations closer to data in the form of controllers near memory, I/O and disks has been discussed for decades. Intelligent disk processing dates back to the late 1970s and early 1980s (DeWitt & Hawthorn 1981; Hurson et al. 1989) where database machines included a central host processor with disk processing techniques such as processor per head, track or disk
or a multiprocessor disk cache arrangement. The database machines performed scan operations closer to the disk. While the scan performance was impressive, database machines did not perform well for more complex operations such as joins, sorts and the narrow performance gains failed to justify the extra cost of the hardware (Boral & DeWitt 1983). These machines fell out of favor due to the limited performance of disks at the time and the complexity of building and programming special purpose hardware that could only handle limited functions. Boral and DeWitt concluded that aggregate storage bandwidth was the principle limitation of database machines.

2.1.2 Intelligent Disks (IDISKs)

Keeton et al. (Keeton, Patterson, & Hellerstein 1998) present an architecture for decision support database servers that utilizes Intelligent Disks (IDISKs). IDISKs use low-cost embedded general-purpose processing, main memory, and high-speed serial communication links on each disk. The IDISK architecture included a high-speed interconnect to overcome the I/O bus bottleneck of conventional systems and provide a scalable I/O subsystem.

2.1.3 Active Disks in the 90s

Riedel et al. (Riedel, Gibson, & Faloutsos 1998) proposed a system in the late 1990s, that they call Active Disks, which takes advantage of the processing power on individual disk drives to run application-level code. Riedel et al. showed that, by aggregating the bandwidth and computing capabilities of ten (emulated) HDDs, measured performance improves more than $2.2\times$, $2.2\times$ and $2.8\times$ for workloads like nearest-neighbor search, frequent set mining and image registration, respectively.

Acharya et al. (Acharya, Uysal, & Saltz 1998) evaluate active disks architectures, and propose a stream-based programming model which allows disklets (disk-resident code of applications) to be executed efficiently and safely on the processors embedded in the disk drives. Disklets take streams as inputs and generate streams as output. Files (and ranges of files) are represented as
streams. The authors present algorithms from three application domains - relational database processing (select, group-by, sort, datacube), image processing (image convolution) and satellite data processing (generating composite satellite images). They compare the performance of conventional disk and active disk by varying the system parameters and scalability experiments were done by varying the number of disks in each configuration. The results indicated that for all the algorithms, active disks outperformed conventional disks by a factor of 1x-3x for 4 disk configurations and 3x-30x for 32 disk configurations.

By using the concept of active disks, and representing files as objects, Lim et al. (Lim et al. 2001) propose an Active Disk File System (ADFS) where many of the responsibilities of a traditional central file server, such as authentication, pathname lookup, and storage space management, are offloaded to the active disks. These authors suggest that objects (files) can have application-specific operations, which can be run by disk processors, so that only the results are returned to clients. However, they do not provide or implement the proper framework to do that.

None of the previous approaches were implemented on commercially available hardware. Active Disk was designed to exploit internal disk components, but the simulated experiments had to use an additional 64MB bytes of RAM and an external processor to simulate a 4 GB disk.

### 2.1.4 Flash based Solid State Devices (SSDs)

Flash-based Solid State Devices (SSDs), which are emerging as a viable technology for large-scale use in systems supporting data-intensive computing. The recent availability of high-capacity solid-state storage opens the possibility of a new architecture for combining storage and computation, called Active Flash (Boboila et al. 2012). This approach takes advantage of the low latency and architectural flexibility of flash-based storage devices to enable highly distributed, scalable data analysis and post-processing. The internal bandwidth of an SSD often exceed the external bandwidth by factors of 2–4×, and the bandwidth growth over time is expected to grow rapidly due to increased
internal parallelism. These characteristics make them ideal platform for embedded data processing.

Some of the previous works on SDD based in-storage processing (Gao, Ayers, & Kozyrakis 2015; Boboila et al. 2012) has been done using simulators (Sanchez & Kozyrakis 2013; Carbon.SOC.Designer.Plus; Cho et al. 2013) and real SSD product platform (Cho et al. 2013). Samsung 840 Evo, one of the latest high end SSDs, features ARM-based triple cores operating at 400MHz and 1GB of DRAM with eight parallel NAND channels (Samsung.SSD.840.evo).

2.1.4.1 Active Flash: Out-of-Core Data Analytics on Flash Storage Active Flash (Boboila et al. 2012) explores the energy and performance trade-offs in moving computation from host to compute node-local flash storage. On-the-fly data analysis significantly reduces the I/O traffic within the device by analyzing the data while it is still in the controller’s DRAM. Idle-time policy maximizes storage resource utilization by carrying out data analysis during low-activity periods. Also, idle-time scheduling offers flexibility: it permits sustaining the desired (high) application data generation rate, when only part of the data analysis is performed on the controller, and the rest on the host CPU (the Hybrid Active Flash model). Microsoft Research SSD simulator (Agrawal et al. 2008) was used for this work which is based on DiskSim (Bucy et al. 2008).

2.1.4.2 ActiveFlash Discussing Partitioning and pipelining to assign tasks to a set of computing resources. To fully utilize the computing resources in both the host platform and the iSSD, we could assign Map functions to both entities by “partitioning” the job into properly sized sub-jobs. For example, if the input data set is composed of 1,000 files, 400 files could be delegated to the iSSD and the remaining 600 files could be processed by the host CPUs. Finally, the pipelining and the partitioning strategies can be used in combination. In fact, combining both strategies will likely lead to better system performance with higher overall resource utilization.
2.1.4.3 Biscuit: A Framework for Near-Data Processing  Biscuit (Gu et al. 2016) is a novel near-data processing framework designed for modern solid-state drives. Biscuit distributes the data-intensive application to run on the host system and the storage system to reduce the volume of data transferred over the storage network to a host system. Biscuit allows the user to dynamically load user tasks to run on the SSD. Resources needed to run user tasks are allocated at run time. This feature decouples user application development and SSD firmware development, making NDP deployment practical.

2.1.4.4 BlueDBM: Distributed Flash Storage  BlueDBM (Jun et al. 2016) is a new system architecture that has flash-based storage with in-store processing capability and a low-latency high-throughput controller network between storage devices. The BlueDBM architecture includes three major architectural modifications: (1) an in-store processing engine, (2) low-latency storage area network integrated into the storage controller, and (3) file system and flash management software optimized across multiple hardware and software layers. The in-store processing engine has access to four major services: the flash interface, network interface, host interface, and on-storage DRAM buffer.

2.2 Processing in Memory (PIM)

2.2.1 Intelligent RAM (IRAM)

Intelligent RAM (IRAM), merges processing and memory into a single chip to lower memory latency, increase memory bandwidth, and improve energy efficiency as well as to allow more flexible selection of memory size and organization (Patterson et al. 1997). Intelligent RAM effort to develop processor-in-memory technology in which incoming samples are written directly to the memory array for high-bandwidth consumption by bursty software processes. This approach focused mainly on per-node performance and eventually targeted vector-based instruction processing. However, the
popularity of IRAM was limited by the amount of memory on-chip.

### 2.2.2 NDP for In-memory Analytics Frameworks

In-memory Analytics Frameworks for NDP (Gao, Ayers, & Kozyrakis 2015) presents the hardware and software features necessary for efficient and practical NDP for in-memory analytics (MapReduce, graph processing, deep neural networks). By placing simple cores close to memory, this eliminate the energy waste for data movement in these workloads with limited temporal locality. The memory-intensive phases of these workloads execute on NDP cores. The compute-intensive phases execute on the host processor on all systems. This system was simulated using zsim, a fast and accurate simulator for thousand-core systems (Sanchez & Kozyrakis 2013).

### 2.2.3 Active Storage based on HDFS

Active System architecture (Fan, He, & Tan 2016) provides programming interfaces for applications to implement the offload computing algorithm and assignment policy. Dynamic task assignment policy that assigns computing tasks between compute and storage nodes according the feedback information of system environment. Task Scheduler collects and saves the runtime information into the active system. Task assignment policy breaks the original compute task into subtasks and divides these between the compute node and storage node using the feedback information collected by the scheduler. The prototype presented in this work is based on HDFS and provides stream-based programming interfaces for applications.

### 2.2.4 AnalyzeThis: Work-flow aware Storage System

AnalyzeThis (Sim et al.) is a smart, analytics pipeline-aware storage system atop an array of AFEs (Active Flash Elements). AnalyzeThis realizes workflow-awareness by creating a novel analysis data object abstraction, which integrally ties the dataset on the flash device with the analysis
sequence to be performed on the dataset, and the lineage of the dataset. Scheduling, i.e., both data placement and workflow orchestration, can be performed within the storage, in a manner that minimizes unnecessary data movement between the AFEs, and optimizes workflow performance. The results with real-world, complex data analysis workflows indicate that AnalyzeThis can expedite end-to-end runtimes and significantly reduce data movement costs. The analysis workflow job is submitted to the AnalyzeThis storage system. As the input data to be processed becomes available on AnalyzeThis (from experiments, observations or simulations) the workflow that the user has submitted is applied to it. The final processed data, or any of the intermediate data is stored in AnalyzeThis, and may be retained therein, transferred to other repositories that may be available to the user (e.g., archive, PFS), or removed based on lifetime metadata attributes that the user may have associated with the dataset. Thematic to the design of AnalyzeThis is that analysis-awareness be deeply embedded within each layer of the storage system.

2.2.5 Apache SPARK

Apache Spark (Datasets 2012) (Spark ) is a distributed computing system that implements the map-reduce (Dean & Ghemawat 2008) programming model and is attracting wide attention for Big Data processing. Spark achieves high throughput compared to the previous dominating open-source map-reduce implementation Apache Hadoop (Hadoop ) by retaining data in main memory whenever possible and possibly through better implementation of its operations. Like Hadoop, it supports iterative computation and it improves on speed and resource issues by utilizing in-memory computation. The main abstractions used in this project are called Resilient Distributed Datasets (RDD), which store data in-memory and provide fault tolerance without replication (Zaharia et al. 2012). RDDs can be understood as read-only distributed shared memory (Ni 2013). Spark streamlines the learning process through in-memory caching of intermediate results, significantly cutting down on the number of read and write operations necessary.
2.3 Accelerated Storage

The work on Accelerated Storage (Khasymski 2015) features deep integration of the GPU in a distributed parallel file system utilizing a framework that builds on the resources available in the file system and coordinates the workload in such a way that minimizes data movement across the PCIe bus, while exposing data parallelism to maximize the potential for acceleration on the GPU.

2.4 Summary

Several NDP frameworks which enable processing in-storage and in-memory have been reviewed in this section. Data processing frameworks for storage devices such as disk and flash based SSD were discussed. Some implementations were simulated whereas others were using the hardware. Each of these have different ways of handling the challenges of data distribution, data locality and data partitioning.
Chapter 3

AISTOR - A COMPUTE FRAMEWORK FOR SCALABLE DISTRIBUTED BIG DATA PROCESSING

3.1 Introduction

The cost of data-movement is one of the fundamental issues that modern compute systems processing Big Data workloads needs to address. External data I/O speeds have not been increasing as costs have fallen. Moving large chunks of data between storage and compute nodes is still highly inefficient as the storage systems are projected to reach Exabyte scale in the near future (Gantz & Reinsel 2012). The high-cost of data movement imposes energy consumption and degrades the performance of big data applications. Data centers often face network congestion bottleneck resulting from moving large amounts of data over the network channel. Thus, the logical step to address the overhead of data movement is to either increase network speeds or reduce the amount of data communicated over the network.

Increasing the network speed is often associated with new, faster, and more expensive hardware. In modern clusters, the nodes are connected through a low-latency and power-hungry interconnect network such as InfiniBand (Niedzielewski et al. 2020) or Myrinet (Fakih, El Baz, & Kotenko 2020). In such systems, moving data can be more expensive (in terms of the use of system wide resources) than processing data (Koziol & others 2014), and moving the data through an interconnect network is
much more costly (cost as expressed in energy or power efficiency). The monetary cost of networking can be significantly more than the that of storage or compute resources. In fact, the access and transfer of data from storage systems is a huge barrier towards reaching compelling performance and energy efficiency.

In this chapter, the idea of reducing the data movement by processing data directly in the storage devices (ISP) is re-examined. ISP, also known as Active Storage (Riedel et al. 2001) and Computational Storage Devices (CSD), aims to expose computational elements within the storage infrastructure to perform general-purpose computation on data. ISP minimizes data movement by moving the processing inside the storage devices so that data does not have to leave the storage device to be processed. This chapter presents the design and implementation of AiSTOR, a compute framework on an Active Storage platform, which incorporates a software stack and a dedicated multi-core processor for in-storage processing. AiSTOR utilizes the processing power of storage devices by using an array of Active Drive\textsuperscript{T\textregistered}M Devices (ADs) to significantly reduce the bandwidth requirement on the network. The performance and scalability of AiSTOR for distributed processing of Big Data workloads are evaluated. This chapter concludes by discussing a comparative study of other existing data-centric approaches.

3.2 Technological Background

The emergence of Ethernet enabled disk drives and Object-based Storage Devices (OSD) (Gibson et al. 1997; Mesnier, Ganger, & Riedel 2003) have enabled harnessing the processing power of storage devices. The concept of OSD and active disks were merged to create intelligent storage devices directly attached to a network (Du 2005). This section gives an overview of Ethernet-based storage devices and object-based storage devices.
<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Processor</th>
<th>Memory</th>
<th>OS</th>
<th>Disk/HDD</th>
<th>Ethernet</th>
<th>Platform</th>
<th>SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seagate</td>
<td>ARM 32-bit 1.33 GHz, dual core</td>
<td>1GB</td>
<td>Linux OS K/V Store</td>
<td>8TB</td>
<td>One Active and one stand-by links</td>
<td>Closed Platform API/LLVM</td>
<td>N/A</td>
</tr>
<tr>
<td>HGST</td>
<td>ARM 32-bit 1 GHz, 1-core</td>
<td>2 GB DD3</td>
<td>Embedded Linux OS</td>
<td>4TB/8TB</td>
<td>One Active link</td>
<td>Open Platform run applications</td>
<td>N/A</td>
</tr>
<tr>
<td>Toshiba</td>
<td>ARM 64-bit quad-core</td>
<td>N/A</td>
<td>K/V Store Linux OS</td>
<td>4TB 2.5&quot; HDD</td>
<td>Two Ethernet links</td>
<td>Same as Kinetic Drive</td>
<td>2</td>
</tr>
</tbody>
</table>

### 3.2.1 Ethernet Based Storage Devices

Ethernet Drives are storage devices that can support Software-Defined Storage (SDS). Ethernet drives add a small operating system with CPU, memory and a network interface to the hard drives. They can be attached directly to existing network infrastructure without the need for storage servers.

There are three different Ethernet connected disk drive technologies, summarized in Table 3.1. These drives do not merely use Ethernet as a new connection interface; they move the communication protocols from commands on read-and-write data blocks to a higher level of abstraction. This new technology is intended to simplify storage system software stack, allow drive-to-drive data movement, and reduce significant drive I/O activities from servers.

### 3.2.2 Object Based Storage Devices (OSD)

Object storage was introduced to manage data as objects (Factor et al. 2005). With the rapidly increasing amount of data and storage requirements, Object-based Storage has emerged as a promising storage solution (Mesnier, Ganger, & Riedel 2003). The Object-based Storage Devices have become a key element in some recent parallel file systems (such as Lustre (200 2003) and the Panasas File System (Panasas ), industry products (IBM Storage Tank (IBM.Storage.Tank ), EMC Centera (EMC.Centera ) and other scalable storage systems (Oldfield et al. 2006; Weil et al. 2006). They manage data as objects rather than files or blocks. Object storage is flat structured and has the capability to scale out horizontally. OSD combines the advantages of Network Attached Storage...
The instantiation of ADs that were used for this work is the second generation Active Drives which is the latest class of Active Drives from Seagate. This is a Kinetic drive (Hughes 2014; Shetty) with a specially modified software and firmware. The Seagate Kinetic Open Storage platform and its developer tools add two key elements to the traditional hard drives: Object Storage Protocol and Ethernet connection. Seagate Kinetic Open Storage platform (Callahan) replaces the primitive block I/O interface with a key/value API and the traditional SAS or SATA interfaces with a pair of Gigabit Ethernet ports targeting the cloud environment. This enables better performance, scalability and simplicity while significantly reducing total overall costs.

Figure 3.1a is a simplified comparison of the hardware and software stack of a traditional storage system and that of a Kinetic system. In a traditional system, the transit path from application to storage requires multiple layers of manipulation from databases, down through POSIX interfaces, file systems, volume managers and drivers. Information passes over Ethernet, through fiber channel, into RAID controllers, SAS expanders and SATA host bus adapters. The Kinetic Open Storage
Table 3.2. Characteristics of ADs

<table>
<thead>
<tr>
<th>Processor</th>
<th>32-bit dual core ARM Cortex A9 @ 1.33 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>DDR (666 MHz) 1GB RAM per device</td>
</tr>
<tr>
<td></td>
<td>1MB L2 cache, 32KB L1 cache</td>
</tr>
<tr>
<td>Storage</td>
<td>5900 rpm 8TB Kinetic Object Storage with Ethernet interface</td>
</tr>
</tbody>
</table>

platform differs from conventional architectures by providing direct access to the storage devices via Ethernet connection. This enables massive scalability and location independence. The use of Ethernet allows the storage system to be expanded without the need to install a separate Storage Area Network. Each Kinetic Drive has two Ethernet ports (1Gb/s port each) to transfer data to the clients or other drives using TCP/IP over Ethernet.

Figure 3.1b shows how the compute cluster is connected with the active storage cluster with ADs. Each AD (Kinetic Drive with the modified firmware) has an extra ARM processor (32-bit Dual Core Cortex A9 Marvell Armada 382) and an extra RAM chip included on the Printed Circuit Board Assembly (PCBA). Each core has an L1 cache that is split into data and instruction caches each of size 32KB. It also contains a 1MB L2 cache, which is shared between the cores. The ARM processor is connected to the outward-facing Ethernet interface, and it is this processor that processes Ethernet packets from the interface. This chip also runs a small Linux kernel, which receives the Ethernet packets. The Linux kernel runs the Kinetic software, which handles all Kinetic commands. The ARM processor is a fully-fledged application processor so that it can run the Linux kernel and the Kinetic software. Because it is an application processor, it can run arbitrary user code as well. To enable the execution of arbitrary code, the most natural path forward was to modify the Kinetic Protocol and the Kinetic software that runs on Kinetic Drives. These modifications allow users to load, manage, and execute code on drives. Table 3.2 summarizes ADs processor specifications.
3.3 Design and Implementation

This section explains the design and implementation of the AiSTOR framework. The input data, algorithms and how the applications are partitioned between the active storage and compute node are detailed in this section.

3.3.1 AiSTOR Architecture

A simple data-centric processing system consists of at least two components: a host system and an ISP device, also known as a computational storage device. AiSTOR is an in-storage compute framework which provides an environment and programming interface to coordinate the host-side and ISP tasks. Figure 3.2 shows the overall AiSTOR architecture. The host is attached to the ISP devices via an Ethernet switch. The host node (compute node) acts as a coordinator by creating/managing tasks and the associated inputs and outputs. As depicted in Figure 3.2, the host interacts with the ISP device through a command handler. The commands for reading, writing and managing the objects and the offloaded application are managed by the command handler. The ISP devices used for this work are the second generation ADs which extend the Kinetic protocol to allow for arbitrary program execution. The host-side library is responsible for distributing the objects to each of the

Fig. 3.2. AiSTOR - a data-centric compute architecture
Table 3.3. Summary of AD Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Install</td>
<td>Install an applet in a drive(s)</td>
</tr>
<tr>
<td>Uninstall</td>
<td>Uninstall an applet in a drive(s)</td>
</tr>
<tr>
<td>Update</td>
<td>Update an applet in a drive(s)</td>
</tr>
<tr>
<td>Execute</td>
<td>Execute an applet in a drive(s)</td>
</tr>
<tr>
<td>Debug</td>
<td>Debugs the applet on a drive(s)</td>
</tr>
<tr>
<td>Abort</td>
<td>Abort an applet in a drive(s)</td>
</tr>
<tr>
<td>Get Status</td>
<td>Monitor the processing status of an applet</td>
</tr>
</tbody>
</table>

ISP devices. The program that runs on the host triggers the execution of the application functions offloaded to the devices, it waits for the completion of tasks and gathers the results from all the devices. Thus, the host acts as a coordinator which creates and manages the tasks on ISP devices. In addition to coordinating the tasks on ISP devices, the host-side program also does the required computations and final merging of results gathered from ISP devices.

### 3.3.2 Off-loadable executable (Applet)

The applet represents the offloaded application functions and interacts with the objects to run the applications. At the end of execution, the applet send the results of the computation to the host device. Table 3.3 lists all the ADs commands. Each item on the command list operates on arbitrary user code which is referred to as an applet. With this set of commands, ADs users are able to install their applets in the storage device, execute them, monitor their processing statuses, get execution results, abort processes, debug their applets, update, and uninstall. These ISP devices are capable of supporting multiple threads and sessions. While the Kinetic API abstracts away most of the internal details of the drives, there are set limits to certain parameters. These limits are listed in Table 3.4.

An applet is similar to an object which is identified by the object ID. The current version of ADs support applets be written in C or a cross-platform language such as Java. To provide a simple and convenient way for users to apply several application functions on the data object, a set
of commonly used functions are included in an applet. This facilitates ease of programming since the user can easily access this functionality by passing the function name and the required attributes. It also enables chaining multiple functions on a single object.

3.3.3 Resource Monitoring

AiSTOR monitors the CPU utilization in the host and ISP devices. Resource monitoring on the ISP device is done by loading an executable to fetch the status of the offloaded application while it is running on the storage device. It samples this information every few seconds, the measurement interval can be varied by specifying a parameter. The CPU utilization is recorded as a number that ranges from 0 to 1. These metrics such as CPU utilization and the temperature of the processor cores could be useful for load balancing or workload characterization.

3.3.4 Safety Considerations

To ensure the security of the overall system, two aspects of security are considered: bad code in the applet and isolating the data between users. Access control lists allows restricting objects on a per user basis. This prevents the users from accessing the data objects that belongs to different users. Permissions set for each operation allows to specify read, write or execute privileges for users.
The execution of an applet can be restricted to use only the objects for a specific user. This ensures that a user cannot access the data objects that belong to another user through applets.

3.3.5 Input Data

MODIS (MODe-rate-resolution Imaging Spectrometer) Surface Reflectance data was used as input data for this evaluation. This data is available in the Hierarchical Data Format (HDF) (Group & others 2000). Since the maximum value size (as in Table 3.4) in ADs is 1MB, the input file was broken down to 1MB chunks and was transferred to the drives.

3.3.6 Data Allocation

In a Big Data application using a large dataset, data allocation becomes a bottleneck, i.e. deciding which storage device to allocate a given key/value pair to. For a smaller dataset, it is easier to have a metadata file to maintain a small-sized indexing table which contains the IP address of device and the corresponding key(s). The disadvantage of maintaining a metadata file is as the data grows larger, the metadata file gets bigger. For a terabyte-sized dataset, the metadata file could end up being a performance bottleneck. To avoid this, we used a deterministic approach of data allocation as explained below.

Each key is assigned a prefix value. The value of prefix is an integer starting at 0 and is incremented for each new 1MB data chunk. The range of prefix is determined by the total number of 1MB chunks for each the input file. Key-value pairs are allocated to each of the ADs based on the value of prefix. All the devices in a storage cluster are assigned numbers starting from 0. For example, if there are n devices, each device would be numbered from 0 to (n-1). A modulus operation of the prefix and the total number of storage devices would give a number which is mapped to the corresponding storage device. In a storage cluster with 24 devices, the key/value pair with prefix 0 is stored in device 0 as shown below:
0 % 24 => device 0
1 % 24 => device 1
...
82 % 24 => device 10
83 % 24 => device 11

We used a deterministic data allocation mechanism to store the key/value pairs ADs. This scheme has the following advantages: (1) It distributes the keys uniformly across all the ADs. (2) It eliminates the need of a metadata file since knowing prefix range for the keys and the device number for each device in a storage cluster, the key distribution for each device could be determined.

3.4 Evaluation

3.4.1 Map Reduce on ISP Devices

The MapReduce programming model facilitates parallel programming and computation close to data by borrowing two concepts from functional programming: the Map function, which processes input data and produces key/value pairs as output, and the Reduce function, which processes the key/value pairs from the Map output and produces final key/value results as output. With this design many algorithms can be adapted to be run in the MapReduce programming model (Chu et al. 2007). Such parallelization achieves quicker computation than other methods by attempting to reduce the amount of data that is moved between computers.

In this use case, we will detail how the MapReduce programming model was applied to ADs to perform general-purpose computing directly on the storage device. This section details the MapReduce implementation on ADs, and the results indicate a significant reduction in the amount of data leaving the ISP devices when performing several widely used data analysis applications.
3.4.1.1 Design and Implementation  The MapReduce algorithm is converted to work with ADs by linking the reduce function into an executable to be run on the host compute node. In addition to the reduce function, the executable on the host consists of two managerial tasks. First, it is responsible for triggering the execution of map function on each ADs. Second, waits for completion for the map tasks running on the ADs and gathers the output of all map functions. The results are sorted and grouped by key. Similar as in the case of Hadoop MapReduce, the reduce executable on the host operates on the key/value pairs returned by the map function to obtain the final results. The map function is compiled as an applet and uploaded on each of the ADs ahead of running the application.

The MapReduce execution flow on ADs are as shown in Figure 3.3. The MapReduce task is triggered by the execution of the reduce function on the host or compute node. The first managerial task part of the reduce executable identifies all ISP devices which are involved in the MapReduce computation. Subsequently, this starts up the Map applets on each storage device. It then monitors the applets for completion. After completion, the second managerial code gathers the output of all
Map applets and feeds them as input to the reduce function. The output of the reduce function is then written as the final result.

### 3.4.1.2 Applications

We used four widely used data analysis applications: Database selection, Data aggregation (average) with group by, Histogram and Gridding. These applications are representative of real world data analysis applications and are widely used in various fields. Table 3.5 lists the applications that are characterized as CPU or I/O bound. The size of the input data for all the applications varied from 1 - 12 TB. The table also indicates the amount of data that was retrieved by the applications. These applications were implemented using the MapReduce framework on ADs and the performance was compared with Apache Hadoop and Spark based implementation.

For the implementation on ADs, each application was partitioned between a host-resident component, which runs on the compute node and an applet, which runs on the active storage devices. Below is a brief description of each application and how it was partitioned between the compute node and ADs.

1. **Database Selection**: A select operation is one of the most important and widely used query in the database system. It matches a given search condition on one or more attributes and returns those records which match the specified condition. The search condition is compiled as the map applet and it returns all the records which match the criteria to the reduce executable running on the compute node. The reduce function collects the output of all the map applets on each storage device and merges the records to create the final output. The map applet performs the select operation and the final merging takes place at the compute node.

2. **Data aggregation with group-by**: Data aggregation gathers information to express a summary for statistical data analysis or for extracting meaningful conclusions from the data. This function returns a single result for the values of the chosen attributes which represents a summary value. The aggregate functions are widely used along with grouping results based on one or several attributes. Examples of aggregate functions are minimum, maximum, average, count and sum. It
uses the split-apply-combine strategy for data analysis. The split and combine phase are done in the map phase which runs on all of the ADs. The reduce executable on the compute node collects the output from all the map applets and performs the final apply phase.

(2) **Histogram:** Histogram is an accurate representation of the distribution of numerical data by grouping the number of data points that lie within a range of values (known as classes or bins). It is used widely in the areas of statistics, image processing, hydrology etc. The map applet computes the count of all the data points that fall in a certain range. It accumulates the count for each bin. The reduce function on the compute node collects the results from all the storage devices and performs the final aggregation.

(2) **Gridding:** Gridding converts any geo-referenced dataset of land, atmosphere or ocean at any level of processing from its non-uniform instrument domain to a uniform sparse-time domain [55]. This allows any geophysical property to be projected to a space-time grid. The grid size, time period and other grid properties are user defined. One of the main advantages of gridding is data reduction. This algorithm consists of two phases. In the first phase, each measurement is assigned a grid ID which is computed based on the grid resolution and time period. Measurements are clustered into nearest neighbor grid cells according to the pre-defined grid resolution. The second phase consists of statistical data reduction where daily, monthly or yearly statistics are calculated for all the data points in each grid cell or nearest-neighbor set. Data points are accumulated over a certain period and a time-aggregated statistical grid is calculated. This application is used in the areas of Remote Sensing for Earth Science and Satellite Image/data processing.

To convert this algorithm to the MapReduce model on ADs, a grid cell ID is computed in the map applet based on the geo-location coordinates for data measurement. For each grid cell two values are computed: the first which is a summation of all measurements for this grid cell and a count of measurements which fell into this grid cell. The data returned from each map applet consists of the grid ID, summation and count values for each grid cell. The output key is the grid cell ID and
the output value is a concatenation of the summation and count values. On the compute node, the reduce function collects the key/value pairs from all map applets which are grouped by the key. The reduce function performs the accumulation and the time-aggregation for all the grid IDs in the final result. For this study, we mapped the data measurements to a global grid of resolution $1^\circ \times 1^\circ$.

3.4.1.3 Input Data MODIS (MODerate-resolution Imaging Spectrometer) Surface Reflectance data was used as input data for this evaluation. This data is available in the Hierarchical Data Format (HDF) (Group & others 2000). Since the maximum value size (as in Table 3.4) in ADs is 1MB, the input file was broken down to 1MB chunks and was transferred to the drives.

3.4.2 Testbed Configuration

This section presents the characteristics of the system and the configuration ADs cluster used for this evaluation. We compare the execution of the algorithms on ISP devices with the Hadoop and Spark based implementations.

The ADs cluster was set up with 24 Active Drives, split across two modified SuperMicro SAS chassis. The SAS backplane in each chassis was replaced with a Seagate-designed Ethernet backplane, known as BlueSky. Each Active Drive was a 5,900RPM Kinetic drive, with 8TB of storage per drive and two Ethernet connections of 1Gb/s. The total storage capacity of this cluster is 192TB. Each AD had dual core ARM processors as compute resources. The ARM cores on these drives are 32-bit ARMv7 Dual Core Cortex A9 Marvell Armada 382 running at 1.33GHz. Thus, the BlueSky Active Storage cluster had 24 dual core ARM processors as compute resources.

Hadoop cluster was set up with 24 compute nodes, each with a local HDD for storage. 4 Hadoop nodes had Western Digital 250GB HDDs, 12 nodes had Western Digital 2TB HDDs. All Hadoop local storage HDDs were the same speed at 7,200RPM. The total storage capacity of the Hadoop cluster was 25TB. The Hadoop cluster had no compute resources on the HHDs, because it used standard HDDs. Each compute node on the Hadoop cluster contains a dual-processor motherboard.
setup which uses 2 quad-core Intel Xeon X5560s, running at 2.8GHz.

The communication channel for the AD was via Ethernet, whereas the Hadoop cluster was connected via SATA. The two AD chassis were connected to each other via 10GB Ethernet link. The Hadoop nodes were connected to each other via a 10GB Ethernet link.

Apache Spark ecosystem was the part of same Hadoop cluster and therefore the hardware resource summary remains the same as that of Hadoop. Spark was set up to run in a standalone mode with HDFS. This evaluation used stable versions of Hadoop (2.2.0) and Spark (1.6.1) with HDFS (2.2.0).

Table 3.5. Workload Characteristics

<table>
<thead>
<tr>
<th>Application</th>
<th>Characterization</th>
<th>Input Data Size</th>
<th>Output Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Select</td>
<td>I/O-bound</td>
<td>1-12 TB</td>
<td>270MB - 4.4 GB</td>
</tr>
<tr>
<td>Data aggregation with group-by</td>
<td>CPU-bound</td>
<td>1-12TB</td>
<td>17 - 204 B</td>
</tr>
<tr>
<td>Histogram</td>
<td>CPU-bound</td>
<td>1-12TB</td>
<td>106 - 284 B</td>
</tr>
<tr>
<td>Gridding</td>
<td>CPU-bound</td>
<td>1-12 TB</td>
<td>2.3 - 15.8 MB</td>
</tr>
</tbody>
</table>

3.4.3 Results

We evaluate the performance of the applications on the frameworks used for this study. The dataset size was varied from 1-12TB. To analyze the scalability, we performed the experiments by varying the number of nodes (ISP devices). The applications used for this evaluation which, are summarized in Table 3.5, were characterized as CPU bound and I/O bound. The output data size in the Table 3.5 indicates the amount of data that was retrieved as the final output.

Time measurements were taken using the Linux time utility and the real (wall clock) time was used for performance measurements. The maximum size of the object (value) supported by AD is 1MB. However, the block size in Hadoop/Spark is 128 MB. The applets run in parallel on all the ISP
devices. For comparison with Hadoop, 24 datanodes were configured to match this. In the Spark cluster, the operations parallelism was set to the total number of cores in the cluster.

Figure 3.4 shows the performance summary of applications on AD cluster. The executions times for the applications as the dataset size was varied from 1-12TB are indicated in Figure 3.4a. The performance scales linearly with the number of ISP device, as depicted in Figure 3.4b. The speedup of application execution time is shown in Figure 3.4b for different number of devices in the cluster. For this experiment, each application was run by varying the number of ISP devices attached to a single host via Ethernet. The Gridding results show that, with 24 ISP devices, the execution time was accelerated by a factor of 12.

3.4.3.1 ADs vs. Hadoop: Figure 3.5 shows the execution times for all applications on the AD cluster and Hadoop/Spark cluster. This demonstrates the performance of ADs compared with Hadoop and Spark implementations. In all the cases, ADs outperform Hadoop by up to 18%. For the 1TB dataset, the MapReduce on Hadoop cluster was slower that that on ADs by up to 11%.

Fig. 3.4. Performance Summary of applications on ADs cluster
percent difference slightly reduces for larger datasets. With 12 TB of data, the difference in runtimes on Hadoop and ADs is about 7%. Gridding is the application which shows the smallest performance gap when comparing ADs with Hadoop based implementation. One of the main configuration differences between ADs and Hadoop is the block size. We chose to keep the block size in Hadoop as 128 MB instead of splitting the input data files into 1MB chunks to match the configuration on ADs. Smaller chunk sizes limits the performance on HDFS by increasing the number of seek operations to retrieve large number of small files which would result in an inefficient data access pattern. HDFS is limited in its ability to support random reading of small files due to its high capacity design. In Hadoop, each record is parsed and broken down to key/value pairs. Frequent checkpointing and shuffling of intermediate results further limit the processing in Hadoop. In ADs, the output of the applet for each input chunk is written to a buffer. When the buffer is full, it gets written to the local storage. Finally, when the applet running on the ADs parses all the input chunks on each device, the output is transferred back to the compute node. The executable on the host waits for completion of the applets, copies over the output from all the devices and performs the necessary computations. MapReduce on Hadoop cluster require a lot of I/O operations and this limits its performance.

3.4.3.2 ADs vs. Spark: Spark outperforms the MapReduce implementations on ADs and Hadoop by up to 21% and 40% respectively. Figure 3.5 shows the execution times of the applications using Spark by varying the size of the dataset size from 1TB to 12TB. For all the applications with 1TB of data, the applications on ADs are slower than on Spark based in-memory processing by up to 21%. However, as the data grows larger in size, the performance gap reduces to up to 8%. Although there is a performance gap between Spark and ADs, the time required to move a terabyte-sized dataset from a datacenter or to download or stream the data to a Spark-enabled compute cluster would exceed this performance gap. An added advantage of using the ADs is the idle time that the data is stored in a storage device could be utilized to perform computations. Apache Spark
Select execution times: (1) AD outperforms Hadoop by upto 13%. (2) Spark outperforms AD by up to 21%, which reduces to 8% with 12TB datasets

Data Aggregation times: (1) AD outperforms Hadoop by a margin of 9-14%. (2) Spark outperforms AD by up to 21% and Hadoop by up to 38%

Histogram execution times: (1) AD outperforms Hadoop by 16%. (2) Spark outperforms AD by up to 21%

Gridding execution times: (1) AD outperforms Hadoop by 7-18%. (2) Spark outperforms AD by up to 16% and Hadoop by 38%

Fig. 3.5. Performance Summary of MapReduce on Hadoop, Spark and AD cluster
is relatively faster, because it caches the input data in memory by using the concept of Resilient Distributed Datasets (RDDs). As a result it accesses the data from RAM instead of the disk. Unlike the Hadoop based approach, little time is spent in moving the data or processes in and out of the disk, thereby reducing the I/Os.

Table 3.6. Summary of Previous In-Storage Computation

<table>
<thead>
<tr>
<th>Architecture ISP Device</th>
<th>Target Storage</th>
<th>Programming Model</th>
<th>File system support</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gu (Gu et al. 2016)</td>
<td>ARM Cortex-R7 SSD</td>
<td>Custom API</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Jun (Jun et al. 2016)</td>
<td>FPGA-SSD SSD</td>
<td>RTL-Design</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Kim (Kim et al. 2016)</td>
<td>ARM-A9 Simulator SSD</td>
<td>Application specific code</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Tiwari (Tiwari et al. 2013)</td>
<td>ARM-A9 OpenSSD</td>
<td>–</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Abbani (Abbani et al. 2011)</td>
<td>FPGA-SSD microprocessor SSD</td>
<td>RTL Code</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>AiSTOR</td>
<td>ARM Cortex-A9 HDD</td>
<td>C, Java, Python</td>
<td>Yes</td>
<td>Linux Kernel</td>
</tr>
</tbody>
</table>

3.5 Comparison with other Frameworks

This section compares the AiSTOR framework with other existing frameworks for in-storage processing. Since each framework is based on a different hardware and software platform, we focus on discussing the features of the frameworks and how well it adapts to the applications. Table 3.6 shows a summary of the critical features of various in-storage processing frameworks.
Xie et al. (Xie et al. 2016) proposed an active storage framework for object-storage devices based on the T10 OSD standard. In this work, they associate a function object with one or several OSD objects by storing the function-object’s ID and parameters within the OSD-object’s attribute. Although this offers the user-level flexibility and transparency, this approach could incur a performance overhead. Based on whether this information is embedded into the object’s key field, this could add to the overhead of reading large key sizes. Further, it would add to the management overhead of modifying the object attributes for each object. These overheads are less significant for a small amount of data but for larger data sets, this could end up to be a significant challenge.

BlueDBM (Jun, Liu, & Fleming 2014) is a scalable architecture that has flash-based storage with and FPGA-based in-store processing capability. The most important advantage of using FPGA accelerators is power efficiency. It lacks the flexibility to support a wide range of applications. FPGA-based architectures are advantageous for specific applications. A newer version (Jun et al. 2016) of this framework is a scalable system which supports more storage nodes and applications. The extra effort to generate RTL code to reconfigure FPGA for each application is overcome by running in-storage applications as instruction-based executables on a microprocessor (Abbani et al. 2011). This approach incorporates a simple operating system with drive-resident utility programs on a soft microprocessor in an FPGA. Exploiting the embedded processors in SSD controllers (Kim et al. 2016; Do et al. 2013; Tiwari et al. 2013) for data processing tasks has several disadvantages. This approach impacts the read/write performance of user applications since the user tasks can interfere with flash management tasks. The primary limitations of the previous works that are addressed in this work is:

- A dedicated general purpose processor in the storage device for in-situ processing, which does not degrade the performance of the storage device.

- Incorporates a simplified storage system software stack, that supports a scalable in-storage processing architecture
• Support a Linux kernel in the storage device.

3.6 Discussion

In this work, we investigated an in-storage processing system and we have demonstrated that it can effectively be used to augment the performance of conventional compute systems. The following are the observations based on the results and analysis.

3.6.1 Cost of an In-Storage Processing Framework

From the survey of published research in the area of In-Storage processing, the choice of data storage in the existing frameworks primarily utilize two types of storage media: SSD and HDD. Some architectures include FPGA accelerators along with the storage device. The design of an in-storage processing framework requires careful consideration of the choice of storage media, different programming models and their implementation costs on the chosen platform. The main performance gap between SSDs and HDDs is dependent on the concurrent data requests (Hruska ). Although SSDs have numerous advantages over HDDs, the latter is still the choice for large volume storage. The cost per TB for SSDs is far more than that of HDDs. The Gartner forecast (Mellor ) indicates that the percentage difference in the $/GB prices for SSD and HDD is around 9X differential, moving to around 85 per cent following the trend.

Another important aspect of cost is the software. The software costs usually outweigh the hardware costs. Ease of programming is an important factor which determines how flexible the in-storage processing framework proves to be for the end users. The design and programmability for AiSTOR is relatively straightforward. The choice of programming languages and the availability of standard libraries reduces the programming effort of the user. The amount of user code required in the applet (offloaded executable) barely exceeds a few hundred lines, given the complexity of the applications.
3.6.2 Is In-Storage Processing for all?

All applications do not significantly benefit from In-storage processing. Processing the user applications inside storage units without sending data to the host processor seems appealing when:

(i) all or most of the data exists in the storage device. It does not make sense to move the processing to an in-storage processing device when the data does not already exist in storage. The application must have data processing tasks that uses the stored data. (ii) The data processing tasks with light-weight compute tasks (with fewer instructions to processes data units) could benefit from in-storage processing. The applications could be partitioned to offload appropriate tasks to the storage devices to maximize the advantages of this approach.

Through the experiments, we demonstrated cases where the entire application was offloaded to storage (database select) and where part of the application was done in the storage device (aggregation, histogram, gridding). We were able to achieve a significant reduction in data transfer and speedup as we scaled up the number of processing devices.

3.7 Conclusion

Evolving technologies are changing the possibilities to run cost-effective storage infrastructures. Object storage and Ethernet drive technology open up new ways to build storage architecture, which addresses the challenges of data management in a compute-centric system. We have presented an active storage architecture to take advantage of the storage parallelism of the storage devices. The primary goal of ADs is to bring a significant reduction in the data communicated over the network, thereby reducing the network congestion in a data center environment handling large amounts of data.

AiSTOR is an in-storage general purpose computation framework on ADs featuring a dedicated multi-core application processor. The dedicated hardware resource provides in-storage processing capability to run applications without degrading the performance of common storage device operations.
Although the processing power of the disk processor is considerably less than that on the compute nodes, the algorithms can take advantage of the aggregate compute power of large disk arrays. Applications can be effectively partitioned to take advantage of this storage parallelism and can have comparable total compute power with those running only on the compute nodes. Applications such as a database select queries running directly on the AD return a small fraction of the relevant data back to the host. This can significantly reduce the network traffic. Data aggregation operations (such as count, minimum, maximum, average) benefit from the combined compute power of the disks and perform simple operations directly on the data. By doing this, these algorithms offload the host node and greatly increase the total compute power available to them. Other applications that could have been good candidates for ISP devices are those which would have minimal computing and return a fraction of data back to the host.

One of the main problem is that good performance requires both good data locality and good resource utilization. A characteristic of big data analytics is that the amount of data that is processed is typically large in comparison with the amount of computation done on it. In this case, processing can benefit from data locality, which can be achieved by moving the computation close the to data, rather than vice versa. Good utilization of resources means that the data processing is done with maximal parallelization. Both locality and resource utilization are aspects of the programming framework’s runtime system. Requiring the programmer to work explicitly with parallel process creation and process placement is not desirable. Thus, specifying good optimization that would relieve the programmer from low-level, error-prone instrumentation to achieve good performance is essential.

ADs benefit I/O-bound scans in two principle ways: 1) parallelism - massive amounts of data partitioned over many ISP devices allows embarrassingly parallel scans to convert a group of ADs into a programmable parallel-scan database machine. 2) bandwidth reduction - scans that filter data with a high degree of selectivity or compute only summary statistics transfer a very small fraction of
the data size from the ADs to the host. For highly selective scans a group of ISP devices can process
data at the aggregate disk rates in a machine whose interconnect bandwidth was designed for much
less bandwidth-demanding applications.
Chapter 4

PROCESSING IN MEMORY ARCHITECTURE FOR
MEMORY-BOUND COMPUTATIONS

4.1 Introduction

In the last decade, despite the advancements in computer architecture, the performance of general purpose processors has been mainly leveraged by technology scaling and micro-architectural research (Zhislina 2015; Moore & others 1975; Märtin 2014; Hu, Stow, & Xie 2018). However, as technology scaling slows down with an upcoming end of Moore's Law, the recent minor improvements presented by traditional architectures are insufficient to reduce the dependency on technology for further performance improvements (Track, Forbes, & Strawn 2017). Nonetheless, improving process technology, increasing clock frequency, decreasing voltage has not been enough for breaking the walls (memory, bandwidth and power wall) which are the main bottleneck of conventional compute systems.

When the traditional 2D manufacturing process failed to diminish the performance gap between processor power and memory bandwidth, die-stacking technologies were explored to mitigate the effect of the bandwidth wall. The 3D die-stacked memory vertically integrates several memory dies on top of each other and connecting them through dense vias thereby, enabling high-bandwidth and high-capacity memory systems (Hu, Stow, & Xie 2018).
Due to the end of Dennard scaling (Esmaeilzadeh et al. 2011; 2013) and the increased demand for performance, the exascale systems require new architectures to extract performance and minimize energy overhead. Scaling existing architectures to large-scale data-intensive applications is limited by energy and performance losses caused by off-chip memory communication and data movements in the cache hierarchy. A viable approach to achieve such computing capacity consists of avoiding data movements by performing computation where the data resides. This concept is widely known as Processing-in-Memory (PIM) and Near-Data Processing (NDP) (Siegl, Buchty, & Berekovic 2016).

The main motivation for developing the PIM approach is to mitigate the memory wall (Stanley-Marbell, Cabezas, & Luijten 2011), bandwidth wall (Kagi, Goodman, & Burger 1996) and the power wall (Pollack 1999) which is primarily due to the data movement to the main processor unit. The PIM architectures are aimed at reducing data movement between main memory and processors, by placing processing units in close proximity with data.

Processor-in-memory (PIM) prototypes with significant performance improvement were built in 1990s (Draper et al. 2002; Kozyrakis et al. 1997; Sterling & Zima 2002). The cost of fabrication process and inefficient speed and memory capacity trade-off led to the limited success of this approach. After almost two decades of dormancy, research in PIM is regaining traction primarily because of three reasons: First, technological advancements in 3D and 2.5D stacking that blends logic and memory in the same package. Second, moving the computation closer to where the data reside allows for sidestepping the performance and energy bottlenecks due to data movement by circumventing memory-package pin-count limitations. Third, with the advent of modern data-intensive applications in areas like material science, astronomy, health care, etc., calls for newer architectures.

This chapter reviews the limitations of current architectures, provides a brief background on the evolution of memory technology, highlights recent work using 3D-stacking technology. The generic NDP architecture is explained and the NDP controller design to accelerate memory-bound computations are detailed followed by the results and evaluation.
4.2 Limitations of current Architectures

The main restriction faced by modern processor architecture is the preservation of the Von Neumann design. This dependency has contributed to inefficient data movement leading to excess memory operations (Shaafiee, Logeswaran, & Seddon 2017). The current architectural limitations which constrict the performance or energy efficiency on traditional compute systems are discussed in this section.

4.2.1 Memory Wall

The increasing discrepancy between processor and memory access speeds is often referred to as the memory wall (Wulf & McKee 1995). The memory wall causes memory access to be the primary performance bottleneck in most of today’s data-intensive applications. The most prominent attempt to mitigate this ever-increasing performance gap targets increasing the memory hierarchy by improving cache memories. However, with the approaching demise of Moore’s Law and Dennard’s Scaling, cache memory is no longer a viable solution for mitigating the memory wall (Santos et al. 2016; Shahab et al. 2018). Large cache memories have significant area and power requirements and incur latencies similar to the memory wall behavior (Santos et al. 2016; Shahab et al. 2018).

4.2.2 Bandwidth Wall

Due to the increasing number of cores and the trend towards vector instructions (ARM 2019; INTEL 2017 b) current processor architectures put high pressure on the memory systems for accessing large amounts of data. To enable the use of many processor cores and their vector functional units, the newer processors are equipped with multiple channels and memory controllers, introducing parallelism at memory modules level and therefore, providing higher bandwidths (Rahman 2013). The increase in the number of memory channels would in turn increase the resources consumed by multiple data buses and sophisticated memory controllers can conflict with the area, power, and energy constraints. Cache
memories suffer the same restrictions of the area, power, and energy constraints. Hence, depending on applications, cache memories can prove to be a bottleneck on modern designs (Santos et al. 2016; Shahab et al. 2018). These constraints could hamper the exploitation of data-level parallelism, restricting the efficiency of more extensive vector instructions on modern processors, which implement vector units capable of processing 64 Bytes of data (INTEL.2017 b). Table 4.1 shows the Last Level Cache (LLC) main memory, and processing logic requirements in terms of bandwidth in the past decade. A comparison between LLC bandwidth and processor core throughput illustrates the performance gap. This performance gap between main memory and processor (memory wall), is a bottleneck for applications using large data sets.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Year</th>
<th>LLC</th>
<th>Main Memory Channels</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem</td>
<td>2008</td>
<td>213GB/s</td>
<td>42GB/s</td>
<td>425GB/s</td>
</tr>
<tr>
<td>Sandybridge</td>
<td>2011</td>
<td>230GB/s</td>
<td>57GB/s</td>
<td>920GB/s</td>
</tr>
<tr>
<td>SkylakeX</td>
<td>2018</td>
<td>245GB/s</td>
<td>85GB/s</td>
<td>1,944GB/s</td>
</tr>
</tbody>
</table>

Source: (INTEL.2018 ; INTEL.2013 ; INTEL.2008 )

4.2.3 Data Movement

Cache memories widely exploit the temporal locality. However, many applications have low or no temporal locality in the most critical parts (Santos et al. 2016; Shahab et al. 2018). The lack of temporal locality results in continuous access to the main memory, which resembles a streaming-like behavior. Cache misses significantly lower the performance and energy efficiency. Although the prefetch mechanisms try to mitigate cache misses, they prove to be ineffective for sparse or irregular stride access (INTEL.2018 ). Further, this impacts the energy efficiency because of minimal data reuse.
4.2.4 Power Wall

The end of Dennard scaling has made all systems energy-constrained. Due to the constraints of power dissipation (Taylor 2012), a high density of active transistors at the maximum operating frequency is no longer viable. These drawbacks can be witnessed in modern processors that present physical limitations on exploring large widths in vector functional and LOAD/STORE units, cache memory lines, and internal register buses (e.g., Intel’s Skylake processor is inefficient to accommodate many vector units (INTEL.2018)). Consequently, the multi-core approach increases costs and requires complex communication systems to avoid cross-core performance interference due to contention for shared resources in the memory system (Zhao et al. 2013). Although many multi-core processors are applied in environments that demand power efficiency, operating frequencies, and number of active cores, different techniques must be adopted to avoid endangering the hardware and excessive energy consumption. Thus, a compromise between taking advantage of the area and avoiding jeopardizing the available hardware resources is essential to improve the overall efficiency of the system.

Modern compute systems usually centralize the execution of applications in sophisticated units such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs), which leads to the use of area and power budgets in a non-efficient manner. To allow the exploitation of high memory bandwidth, and to improve instruction and data-level parallelism capabilities, it is required to have a non-centralized architecture. PIM can take advantage of 3D-stacked memories and can provide the resources to exploit the available bandwidth on 3D-stacked memories. It avoids unnecessary data movement through cache memory hierarchies by processing data where it resides, which improves the overall efficiency.
4.3 Memory Technology Background

4.3.1 DRAM

Dynamic Random Access Memory (DRAM) stores data in two-dimensional arrays of storage cells, called DRAM arrays. Data is represented in storage cell in the form on electric charge and uses a single transistor-capacitor pair to store each bit. To read a value, the entire row specified by the row address is sent to sense amplifiers, where the small charges get amplified for reading. The specified column is selected and read from the sense amplifiers.

Figure 4.1 shows the internal organization of memory. The arrangement of rows and columns within the DRAM arrays and the connection to the external data pins through the I/O subsystem are shown in the figure. A DRAM bank refers to a set of DRAM arrays that act in unison to allow accessing multiple bits of data. The number of bits accessed with a single command is equal to the number of DRAM arrays in the bank and is referred to as the column width. The banks operate independently. All memory read and write requests pass through the memory controller, where the memory request is translated into one or more DRAM access requests and put on the read queue or write queue. The number of DRAM access requests depend on the location and size of requested
data. The DRAM accesses are processed one at a time; read access requests are serviced before the write requests. For most of the modern systems with on die memory controller, the maximum width of the DDR interface is fundamentally limited by how many I/O pins the CPU die allocates for interconnection with the Double Data Rate (DDR) Dual In-line Memory Modules (DIMMs). Since the number of I/O pins on the CPU die is a scarce resource, the DDR bus width has remained the same for the DDR2, DDR3, and DDR4 class of memories with the extra I/O pins allocated utilized to increase the number of DDR channels supported by the CPU.

4.3.2 3D-stacked Memories

3D-stacked memories have emerged as a feasible solution to tackle the memory wall problem and the little performance-efficiency improvement achieved by traditional commodity DRAMs. The 3D memory vertically integrates multiple DRAM layers on a logic layer connected together using vertical through-silicon vias (TSVs). 3D-stacked memories provide high bandwidth, low latency, and significant energy-efficiency improvements in comparison to traditional DDR (Santos, Alves, & Carro 2015; Pawlowski 2011). The most known industrial products that use the 3D-stacked memory technology are Microns’s Hybrid Memory Cube (HMC) (Consortium & others 2013), AMD and Hynix’s High Bandwidth Memory (HBM) (Lee et al. 2014; Jun et al. 2017), Tezzaron DiRAM (Tezzaron) and Samsung’s DDR4 (Samsung.DDR4).

3D XPoint (Wu et al. 2017) is a memory storage technology jointly developed by Intel and Micron Technology Inc. which is intended to fill the performance gap between DRAM and NAND flash with 1000x faster and 1000x more endurable than NAND flash memory and 10x more storage density compared to conventional memory. Intel Optane SSD (B.Tallis 2017) (Intel 2017a) is a non-volatile memory based on 3D Xpoint technology which is a phase-change memory technology, with a transistor-less, cross-point architecture that positions selectors and memory cells at the intersection of perpendicular wires. Since the 3D XPoint cells can be stacked in three dimensions,
this approach improves the storage density. The primary advantage of this over NAND is the ability to write data at bit level and this enables 3D XPoint to have higher performance and lower power consumption than NAND flash. The 3D XPoint is used as an additional layer of storage between flash and DRAM. This means that the applications that benefit from high speed data access are stored on the flash layer, while applications where the data are accessed less frequently are put on the disk.

### 4.3.3 High Bandwidth Memory

JEDEC’s High Bandwidth Memory uses 3D integration to package a set of DRAMs. It consists of four DRAM dies and one single logic die at the bottom. Each DRAM die consists of 2 channels, where each channel has 1 Gb density with a 128-bit data interface and 8 independent banks (Lee et al. 2016a; 2014). HBM uses through-silicon vias (TSVs) as internal communication busses, which enables far wider interfaces. HBM communicates with memory controller through a 2.5D interposer, which has 1024-bit. The available bandwidth with an 8-channel read operation is 128 GBps at 1.2V.

### 4.3.4 Hybrid Memory Cube

Hybrid Memory Cube (HMC) (Consortium & others 2013) contains either four or eight DRAM dies, and one logic layer stacked and connected by a TSV. Each memory cube contains 32 vaults and each vault controller is functionally independent to operate upon 16 memory banks. The available external bandwidth from all vaults is up to 320 GBps, and it is accessible through multiple serial links, while the internal bandwidth can be up to 500 GBps (Jeddeloh & Keeth 2012). The primary difference between HBM and HMC is that while HBM provides high bandwidth on top of JDEC-based communication, HMC utilizes packet-based interfaces and relies on high internal concurrency.

Advantages of HMC over traditional DRAM are:

**Capacity:** Due to the limits of scaling DRAM density have slowed in recent years. In HMC,
several DRAM dies are stacked up on a cube and can contain 4x or 8x storage in the same package footprint as a DRAM device.

**Bandwidth:** HMC's high bandwidth is because of combining dense TSVs. In addition to the TSVs, each cube has several high speed link which provide high-bandwidth to off-chip processors.

**Parallelism:** Each *vault* in HMC is equivalent to DDRx channels. Since the *vault* is operated independently (with one or more memory banks) there is a high level of parallelism inside a cube. With 16 or 32 vaults per cube, the vault-level-parallelism adds an order of magnitude within a package. As vertical stacking allows a greater number of banks per vault, bank-level parallelism also increases the bandwidth within each vault.

**Efficiency:** HMC is more efficient than traditional DDRx memories because of the close connections between the memory controller and the DRAM devices with short TSV bus. 3D-stacked memories like HMC can be 15x more efficient than an equivalent low power DDR from Micron (Weis et al. 2011).

**Near Memory Processing:** HMC consists of a non-DRAM logic die at the bottom of the stack. The logic die of the HMC is a vault controller and it supports a set of operation instructions. The instructions operate on memory operands and write the result back to the DRAM arrays that follow a read-modify-write sequence.

### 4.4 Related Work

This section presents the recent work exploring processing-in-memory as well as challenges that prevent PIM from being broadly adopted.

**History of PIM** The concept of PIM was proposed in the 90s, where the main motivation was to eliminate or lower the memory wall (Stanley-Marbell, Cabezas, & Luijten 2011), bandwidth wall (Kagi, Goodman, & Burger 1996) and the power wall (Pollack 1999). Multiple studies focus on efforts to integrate processing with main memory (Kogge et al. 1996; Patterson et al. 1997; Hall et al. 1999;
Oskin, Chong, & Sherwood 1998; Kang et al. 1999). The first studies on 2D-integrated PIM were not largely adopted. Advances in 3D integration provide new opportunities for a practical implementation technology for PIM without the technology problems that similar efforts had in the past.

**Processing in the DRAM Module or Memory Controller** This approach explores processing in the vicinity of the DRAM module but not within the DRAM chip (Asghari-Moghaddam et al. 2016; Seshadri et al. 2015) thereby, reducing the cost of 3D-stacking as the DRAM module remains unchanged. This approach offers limited efficiency since it uses lesser internal memory bandwidth as compared to the in-RAM approaches and 3D-stacked accelerators (Hashemi et al. 2016).

**Processing in 3D-stacked Memory** The evolution of Through-Silicon-Vias (TSVs), addresses some of the issues associated with 3D-stacked memory such as thermal dissipation (Sakuma et al. 2008). 3D-stacked PIMs with multicore systems placed into the logic layer (Pugsley et al. 2014; Ahn et al. 2015a; Azarkhish et al. 2016a; Drumond et al. 2017; Scrbak et al. 2017), Single Instruction Multiple Data (SIMD) units (Santos et al. 2017; Oliveira et al. 2017b) and Coarse-Grain Reconfigurable Arrays (CGRA) (Gao & Kozyrakis 2016; Farmahini-Farahani et al. 2014). Despite the advances, the recent 3D-PIM architectures still have to address concerns such as data coherence among host processor and PIM units.

**Processing within the memory chip or memory array** Performing memory and arithmetic operations directly within the memory chip and also the memory array take advantage of architectural properties of memory circuits (Seshadri et al. 2017; Kang et al. 2017; Angizi, He, & Fan 2018; Chi et al. 2016). These approaches take advantage of the architectural properties of memory circuits and add bulk operations to them as new properties to the memory chip. Most work in the literature using this concept rely on the bulk copy, data initialization (Seshadri et al. 2013), bulk bitwise operations (Li et al. 2016) and simple arithmetic operations (Shafiee et al. 2016).

**PIM simulators** Most of the recent work on PIM is based on fully-programmable cores simulated by adjusting constraints of 3D integrated circuits in existing simulators which, takes
Fig. 4.2. NDP architecture with 3D-stacked memory

advantage of execution models and compilers. Zsim (Sanchez & Kozyrakis 2013) is a fast and scalable simulator that supports PIM models (Gao, Ayers, & Kozyrakis 2015; Gao et al. 2017; Song et al. 2018). Zsim provides a collection of lightweight user-level virtualization to ease execution of benchmarks. SiNuca (Alves et al. 2015) is an accurate and validated simulator focused on Non-Uniform Cache Architectures (NUCA) simulation, which is used in several PIM studies (Alves et al. 2016; Santos et al. 2017).

Fixed-function in-memory processing, includes HMC and the works on (Ahn et al. 2015b; Gao, Shen, & Zhuo 2018; Oliveira et al. 2017a) using custom tools. In (Xu et al. 2018), the authors present a PIM simulator that integrates a memory and architecture to provide an interconnection of CPU architectures. (Yang, Hou, & He 2019) presents a PIM architecture for Internet-of-Things applications which simulates both PIM and host processing elements and a tool for estimating power consumption.

The main features of PIM simulation tools are summarized in Table 4.2.
Table 4.2. Summary of PIM Simulators

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Processor Simulation</th>
<th>HMC Model</th>
<th>System Support</th>
<th>PIM Extension</th>
<th>Full-system support</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMC-Sim (Leidel &amp; Chen 2016)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>CasHMC (Jeon &amp; Chung 2016)</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Clapps (Oliveira et al. 2017a)</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Zsim (Sanchez &amp; Kozyrakis 2013)</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SiNuca (Alves et al. 2015)</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>PimSim (Xu et al. 2018)</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gem5-HMC (Azarkhish et al. 2015)</td>
<td>✓</td>
<td>-</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PIM-gem5 (de Lima et al. 2018)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(Yang, Hou, &amp; He 2019)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

4.5 NDP Architecture

The generic NDP architecture used for this work is shown in Figure 4.2. The NDP system consists of a host processor chip connected to multiple memory stacks. The memory stack consists of a 3D die-stacked memory with integrated PIM cores. The 3D memory vertically integrates multiple DRAM layers on a logic layer connected together using vertical through-silicon vias (TSVs). Each DRAM layer contains multiple partitions with each partition having multiple banks. The term vault refers to the vertical partition across a die (as marked in the figure). Each vault consists of an independent vault memory controller within the logic layer that manages memory operations of the vault. Each vault has a compute unit referred to as PIM core, placed in the logic die. PIM core facilitates low-latency memory access because of the high internal memory bandwidth of the TSVs and its proximity to the vault. The integration of PIM cores and memory using 3D stacking is shown.
PIM cores are simple lightweight in-order, single cycle cores similar to ARM Cortex A7 (ARM.Cortex-A7). Each PIM core also has private L1 caches for instructions and data, 32 Kbytes each. Applications that are appropriate to exploit PIM are those which are memory-bound and benefit from the simple PIM cores. The PIM cores have minimal compute power with exclusive access to the data in its coupled vault. Hence, the data-intensive computations can be offloaded to the PIM cores to fully exploit the low-latency memory access and high-internal memory bandwidth of the TSV interconnect.

4.5.1 PIM Software Stack

The PIM software stack is shown in Figure 4.3. The PIM device is attached to the logic layer interconnect through its local interconnect. The PIM module consists of a Scratchpad Memory (SPM), Translation Look-aside Buffer (TLB), Direct Memory Access (DMA) engine and a Memory Management Unit (MMU). PIM accesses the user-space virtual memory using the TLB. The memory copy mechanism from user-space to PIM is done with a virtual pointer passing thereby, reducing offloading overheads, improving scalability and programmability. Paging of user memory is supported by PIM through slices. A slice is a region in memory which consists of one or more contiguous memory pages in virtual and physical memory space. The memory management in PIM is done at the granularity of each slice.

DMA engine is used for bulk data transfers between the DRAM vaults and the SPM. The DMA engine allows multiple transactions by using several DMA resources. DMA is more beneficial for computations on the PIM device that are not localized to a single memory vault. Computations with high data locality and low computational intensity are better when performed in the vicinity of the memory dies.
4.5.2 Program on PIM

The program offloaded to the PIM device (resident code) performs the required NDP tasks. PIM module includes a set of configuration registers which are mapped to the physical address space accessed by the host device. PIM device driver is compatible with the standard accelerators and it provides a low-overhead communication with the API and PIM. The API initiates offloading the PIM computations and coordinating the communication. The host processor dynamically offloads the binary executable and linkable file (elf) to the PIM device (kernel offloading). The user data structures are sent to the PIM device for execution (task offloading). The host monitors the progress and completion of the offloaded task through the PIM API. The NDP device receives commands sent from a host processor. The offloaded task on the PIM device receives the commands from the host-resident program to perform the required task. The host processor writes/reads to the PIM device through memory mapped registers which are mapped in the scratchpad memory.
4.5.3 PIM controller Design

The PIM device consists of a simple in-order processor without cache. Therefore, it requests for one word of data from the DRAM. Every memory access is counted as a separate DRAM access, regardless of the row-locality. The repetitive row activation contribute to data movement. In the open-page DRAM access policy, every activated row is kept open until subsequent memory access activate a different row. However, this does not mitigate the delays associated with data movement. The data transfer from DRAM bank to the memory controller is in units of bursts, which are consecutive columns of data. A burst is often larger than a word. When the NDP core requests for one word of data, the extra columns of data is discarded. This process is highly inefficient when the subsequent requests would need the data from a previous burst.

To address this, a buffer is included in the PIM controller as in Figure 4.4. This buffer 1. acts as a single-block cache within the memory controller 2. cuts down unnecessary data movement 3. buffer size is designed to be the same as DRAM row size. Similar to the cache which consists of the tag memory and data memory, this design modification consists of a buffer which holds the data block and tag register which acts as the tag memory, which holds the tag portion of the memory address which is buffered. When the PIM controller receives a read request, it checks the tag register...
where the upper bits of the requested address are compared with the contents of the tag register. If the address matches (a hit), then the data is read from the data buffer. This reduces the data movement and delay in data access from the DRAM banks. In the event of a miss (i.e. the requested data is not present in the data buffer, the PIM controller initiates DRAM access to fetch the data to the buffer.

4.6 Evaluation

4.6.1 Accelerating KNN on a Smart Memory Cube

K-nearest neighbor (KNN) is a fundamental supervised algorithm for machine leaning and data mining (Muja & Lowe 2014; Nam, Kim, & Nam 2016; Chen et al. 2020; Jan et al. 2008). It is an instance-based or a lazy learning method which performs the learning process at the time when new sample is to be classified as opposed to other learning models where the training data is pre-classified before the new sample is to be classified.

We use the k-nearest neighbor search algorithm (KNN) (Cover & Hart 1967; Xiao & Ding 2012) to investigate how near-memory computation can address the memory bottleneck of KNN algorithm. KNN classification methods are extensively used in the areas of pattern recognition (Shanableh, Assaleh, & Al-Rousan 2007; Geng, Zhan, & Zhou 2005), remote sensing (Frigui & Gader 2008; Song et al. 2016), image processing (Mensink et al. 2013), bioinformatics (Maji 2010; Raymer et al. 2003) and Assistive Technologies (AT) (Kim, Park, & Ghovanloo 2012; Sahadat et al. 2018; Huo & Ghovanloo 2010). The KNN algorithm due to its memory-bound nature, is bottle-necked by data movement, limiting throughput and incurring latency in these applications. This preliminary analysis explores utilizing the NDP-enabled 3D-stacked memory to offload the memory-bound parts of the KNN algorithm.

The KNN algorithm computes the distance between the training samples and test samples in the dataset and returns k-nearest training samples to the test samples (i.e. k nearest neighbors) from
all the known samples. These $k$ samples jointly determine the class of each sample. For distance calculations, we use the Euclidean distance as it is one of the widely used distance metrics in KNN.

**Algorithm 1** KNN Algorithm

Input: $k$

Output: top $k$ closest points

forall $i$ in training data do

  Calculate the distance between new instance and all instances of the training data points

  Sort the calculated distances in ascending order based on distance values

  Select top $k$ rows (lowest distance) and their class labels

  Class of the new instance is the most frequent class of the top $k$ instances

end

The KNN algorithm (Lubis, Lubis, & others 2020) is described in Algorithm 1. KNN first computes the distance between each training sample and the target point. In an $n$-dimensional feature space, the Euclidean distance between the training sample $p = (x_1, ..., x_n)$ and test sample $q = (y_1, ..., y_n)$ is

$$\sqrt{n} \sum_{i=0}^{n} (x_i - y_i)^2$$

The smaller value of distance indicates greater similarity. The algorithm returns $k$ closest samples.

The computational complexity of the search method is $O(nd)$, where $n$ is the size of the training dataset and $d$ is the dimensionality (Wu, Zhang, & Zhang 2005; Wu & Zhang 2003). The performance of the KNN algorithm is limited by the memory bottleneck. The Euclidean distance calculations are not compute-intensive but moving the data samples from memory to the compute device is a bottleneck. The data is used once and discarded whereas the output is just a small number of samples. Performance is further limited by dimensionality (Bennett & Brassard 2020). Because of the simplicity of the KNN algorithm and a small output dataset, KNN is an ideal algorithm for
acceleration with the PIM system thereby substantially reducing the need for data movement.

4.6.2 KNN Characterization

The KNN algorithm classifies the data points based on the assumption that the data points closer to the test point are similar. The success of using the KNN algorithm is dependent on a dense dataset. To state otherwise, the primary limitation of KNN is the curse of dimensionality (Kouiroukidis & Evangelidis 2011b). This means that in a high dimensional space, the distance between nearest and farthest points become nearly the same. Thus, distance calculations cannot differentiate between candidate points. The use of indexing data structures mitigate the limitations of high dimensionality for KNN (Kouiroukidis & Evangelidis 2011a). Using this approach, the KNN search may not find the exact closest neighbors. Instead, it maintains a priority queue of the close neighbors which are most likely and searches branches starting with the most likely candidate points. A timeout constraint determines when the algorithm would return the best matches. Three search strategies were examined to evaluate KNN efficiency.

**Linear Search:** By scanning the entire dataset, this search strategy does not use approximation and requires no data structure. Distance is calculated per query-candidate pair and put in a priority queue. The complexity of this approach is $O(Q \times N)$, where $Q$ is the number of query vectors and $N$ is the number of data vectors in the dataset. Although linear search has the advantage of accuracy, it is slower and considered the worst-case upper bound for searching.

**KD-Trees:** In this, the nearest-neighbor search is approximated by kd-trees and search in parallel. In each tree, top $N$ dimensions are selected. While performing the search, the priority queue is shared across trees to keep track of the already visited branches.

**Priority K-means:** A priority queue and k-means data structure is used to approximate the search. The k-means algorithm is used to construct the initial data structure by clustering the dataset into $k$ clusters. Compared to kd-tree, the decomposition is performed across dimensions.
While searching, the tree is traversed by comparing the distance between the query vector and each cluster centroid. Similar to the other approaches, a priority queue of the closest neighbors is stored as the algorithm traverses the queue starting from the closest center point. Priority queue contains the branches in the order of increasing distance with the query vector. The priority K-means has the disadvantage of poor traversals leading to false identification of neighbors in the clusters. The maximum number of branches to be visited determines the level of approximation.

The distance computations are the most compute-intensive part of the KNN algorithm. In linear search, most of the effective compute time is spent in pairwise distance computations. However, a small fraction of time is spent in sorting the distance values to find the nearest neighbors. While using kd-trees and k-means, the KNN distance computations consists of 89% and 99% of the compute time respectively. Calculating the distance computations are performed during the branch traversal phase and the branch-query for the distance computations are performed when the query hits the leaf nodes of the index. The final distance calculation from the query to the leaf node is done at the leaves.

**k and dimensionality:**

<table>
<thead>
<tr>
<th>Application</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>kNN classifier</td>
<td>1</td>
</tr>
<tr>
<td>Lowe’s ratio test (Snavely, Seitz, &amp; Szeliski 2008)</td>
<td>2</td>
</tr>
<tr>
<td>Document Distances</td>
<td>1-19</td>
</tr>
<tr>
<td>Motion Planning</td>
<td>15</td>
</tr>
<tr>
<td>Image Searching</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 4.3 shows the values of k based on different applications. For a basic KNN classifier, a query is classified by the nearest neighbor. Large values of k makes the classifier immune to noise. Lowe’s ratio test (Snavely, Seitz, & Szeliski 2008) eliminates noisy samples by filtering outliers. It
finds two nearest neighbors with distances $d_1$ and $d_2$, find the ratio of distances and if the ratio is below a threshold (0.6), then it accepts the matches. Other applications such as document retrieval, motion planning and content based search require the retrieval of many neighbors. The vector dimensionality is defined by the feature descriptor used or the feature size of the database and queries. The specific task defines the use of a given descriptor, i.e. to differentiate characters of alphabets, a shape-based feature descriptor is ideal. Table 4.4 lists the feature descriptors based on tasks.

<table>
<thead>
<tr>
<th>Feature Descriptor</th>
<th>Dimensionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURF (Kokila, Sannidhan, &amp; Bhandary 2017)</td>
<td>64</td>
</tr>
<tr>
<td>Word Embeddings (Snavely, Seitz, &amp; Szeliski 2008)</td>
<td>50-100</td>
</tr>
<tr>
<td>SIFT (Kokila, Sannidhan, &amp; Bhandary 2017)</td>
<td>128</td>
</tr>
<tr>
<td>GIST (Singh &amp; Kosecka 2010)</td>
<td>960</td>
</tr>
<tr>
<td>CNN-AlexNet (Abd Almisreb, Jamil, &amp; Din 2018)</td>
<td>4096</td>
</tr>
</tbody>
</table>

**Fig. 4.5.** Mapping KNN algorithm on a PIM system

### 4.6.3 Mapping KNN on a PIM system

The KNN algorithm is memory-bound as explained in the previous section. The memory-bound components of the algorithm are offloaded to the PIM core. Figure 4.5 shows the partition of the
KNN algorithm on PIM core and the host processor and data placement in the PIM system. The training samples are pre-loaded on to the PIM memory and the test samples are loaded to the PIM core scratchpad memory. The computation is initiated by the host processor which communicates the input \( k \) value through the memory-mapped PIM memory. The data samples are loaded from the vault memory banks into the NDP core where the Euclidian distance computations are performed. The results of the distance calculation are then reduced to the local \( k \) closest neighbors and communicated back to the host.

The vault controller communicates with the memory vaults and PIM core. Since the PIM core is paired with vault controllers, data access is localized. This eliminates memory bank conflicts and reduces memory access latency variations from non-uniform memory access. Since the dataset vectors are loaded to the scratchpad memory, the redundant data movement from the memory to the host processing unit is reduced. The Euclidian distance computation consists of subtract and multiply operations followed by accumulation. For top \( k \) points, calculated distance and the corresponding IDs are stored in a queue. Each entry in the queue is compared against the current top \( k \) values in the queue. The entries are sorted by the value of distance and entries with the smallest \( k \) values are maintained at the top of the queue.

The distance computations run entirely on the PIM core-vault pair. The host processor communicates the input parameter \( k \) through the memory mapped portion of the PIM core's scratchpad memory. The output is also communicated back to the host via the memory-mapped region. The data samples are stored in the NDP vault and it is read into the scratchpad memory prior to the distance computations. Figure 4.5 shows the host-PIM interaction for the KNN algorithm.

Reading the data samples into the memory mapped area of scratchpad memory reduces the DRAM accesses that results in delays and power consumption. Due to the simplicity of PIM-core and the resulting incapability to handle sophisticated functionality leads to the instructions reading word-length data. The absence of cache results in every instruction incurring DRAM operations to
access small portions of the data block. Reading a data sample from the vault goes through the following steps: The DRAM row that contains the row is activated and the corresponding columns are selected. The data bits are transferred to PIM-core. These steps with minimal delays are repeated for every word of the data sample being read. Each DRAM row would contain several words of data sample stored in a contiguous manner. The entire block of data sample must be read to the scratchpad memory using DMA transfer to effectively eliminate redundant DRAM row activation.

4.6.4 Performance

For this evaluation, we use the SMCSim (Azarkhish et al. 2016b) simulator which includes the software stack and architecture support for NDP with SMC simulation environment to model an SMC with the host SoC. SMCSim is a high-level simulation that is based on gem5 (Binkert et al. 2011) cycle-accurate, full-system NDP architecture simulator with real hardware constraints. This system uses the HMC concept as the main memory part which includes multiple vaults, a vault controller in each vault (Azarkhish et al. 2016b). Vault controllers are augmented which will be used to handle the atomicity. The main memory is divided into 16 memory vaults, all used as host-accessible main memory and shared with the PIM core. We use the shared and split versions of the memory vaults for this evaluation. The PIM layer is located between the memory controllers and the interconnection layer. Table 4.5 summarizes the configuration of the SMCSim framework.

The PIM core is an ARM-Cortex-A15 core with its own interconnection. An offloadable executable referred to as kernel is offloaded to the PIM core for execution. This system also includes a Scratchpad memory (SPM), Direct Memory Access (DMA) engine, Translation Look-aside buffer (TLB), and Memory management unit (MMU) (Azarkhish et al. 2016b). TLB is included inside the PIM in order to reduce the delay since PIM is far away from a host processor. DMA engine is used for transferring data between DRAM vaults and its SPM. DMA engine and SPM are included in order to reduce the latency of data movement from the DRAM.
Table 4.5. Configuration of the PIM framework

<table>
<thead>
<tr>
<th></th>
<th>Host</th>
<th>PIM Core</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor</strong></td>
<td>ARMv7 Cortex-A15 (8 in-order processors)</td>
<td>1 in-order processor/vault (ARMv7 Cortex-A15)</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>L1 (32kB icache, 64kB dcache) L2 (2MB shared)</td>
<td>SPM 40kB/core</td>
</tr>
<tr>
<td><strong>Vault</strong></td>
<td></td>
<td>Vault 128MB/vault</td>
</tr>
</tbody>
</table>

We use the Iris flower dataset (Dua & Graff 2017) available at the UC Irvine Machine Learning repository. It consists of 150 measurements from three species of Iris, four features per sample (length and width of the sepals and petals). For a given data point, KNN classifies the species of the flower based on the attributes.

![KNN Execution times on host and PIM](image)

**Fig. 4.6.** KNN Execution times on host and PIM

Figure 4.6 shows the execution times for KNN algorithm with varying values of $k$. We show that offloading the KNN on the PIM core speeds up the execution time by upto 1.7x for different $k$ values. We evaluated the NDP architecture for a shared and split PIM core. The default configuration
in the SMCSim simulator (Azarkhish et al. 2016b) is the shared PIM configuration in which, the host-accessible main memory is divided into 16 memory vaults and is shared with a single PIM core. In the split mode, each PIM-core is connected directly to a single memory controller with each NDP vault connected to the PIM-core associated with it. This enables running PIM-tasks in parallel on all the PIM cores. We observe a minimal decrease in execution times for a split PIM configuration. Reading the data-stream into the SPM reduces the redundant DRAM access and this is the main benefit of adopting the NDP architecture.

4.6.4.1 Memory Reads  For further analysis, we compare the KNN implementation on PIM controller design in section 4.5.3 (PIM w/ buffer) and on the generic architecture with both open-page (PIM-open page) and close page (PIM-closed page) row management policies. PIM w/buffer uses the close-page policy as explained in section 4.5.3. We compare these with host-based references.

![Figure 4.7. PIM Operation Throughput](image)

Figure 4.7 shows the operation throughput across varying the number of threads. The PIM-based implementations perform better than the host-based implementations although the host
performs better by up to 24% for smaller number of threads. However, as more threads are used, PIM-open page and PIM w/ buffer show significant improvements. At eight threads, PIM w/ buffer and PIM-open page each have 75% and 52% higher operation throughput than the host. Another important observation is that, the operation throughput of PIM w/ buffer scales better than other implementations with an increasing number of threads.

PIM-open page has 3.7% higher throughput than PIM-closed page and PIM w/ buffer has 16% higher throughput than PIM-open page. The performance difference between these implementations are due to the delays associated with DRAM access. The DRAM delays occur due to i. The time to activate DRAM row, $t_{RCD}$ ii. the time to read the DRAM column, $t_{CL}$ iii. move the burst containing the data to the controller, $t_{BURST}$. PIM-closed page, requires doing three steps for subsequent reads whereas in PIM-open page, since the row activated in the previous read is left open, the next DRAM access requires steps 2 and 3. PIM w/ buffer would cache the previous burst and the additional DRAM access could be avoided.

Figure 4.8 shows the memory read requests at the memory controllers, the number of DRAM
accesses and the number of DRAM row activations. In all the PIM-based implementations, the DRAM bursts are nearly the same as the memory read requests since it reads data one word at a time. The number of DRAM activations is reduced to half with PIM-open page and PIM w/ buffer further reduces the number of DRAM read bursts. The host-based memory read requests to the memory controller originates from the cache misses and it fetches data in the cache block chunk size which are multiples of DRAM burst size. Therefore, the number of read requests are significantly lesser than the DRAM read bursts.

4.7 Summary

Many important classes of emerging AI, machine learning, and data analytics applications operate on very large data sets. The conventional computer-centric architectures are not designed to handle such large-scale data. The performance and energy costs of moving data between memory and CPU dominates the cost of computations for large datasets on a compute-centric architecture. Processing-in-memory is an approach to reduce the data-movement bottleneck by performing some or most of the computation in memory by including processing elements within memory.

The main motivation of this study is to reduce the data movement between CPU and memory by efficient application partition between the host and PIM cores. In this work, we examine the effectiveness of PIM for KNN algorithm and our results indicate upto 1.7x speedup in the execution time. Analysis of the operation throughput across varying the number of threads indicate that the PIM based implementations have upto 75% higher operation throughput compared to the host. The results from analyzing memory read requests at the memory controllers, the number of DRAM accesses and the number of DRAM row activations indicate that the number of DRAM activations are significantly reduced with PIM-based implementation. This work is relevant to other machine learning or computer vision applications that process large datasets where data movement is a limiting factor of performance gains. Further analysis is required for measuring the performance
gains in terms of energy savings. Based on our analysis, KNN algorithm has the potential to be further accelerated with compute-capable memory using specialized near-memory accelerators.
Chapter 5

COHERENT ACCELERATOR PROCESSOR INTERFACE FOR HETEROGENEOUS STORAGE AND MEMORY DEVICES

5.1 Introduction

Emerging needs as we approach for exascale computing bring new challenges related to heterogeneous computing which requires intensive analysis of massive and dynamic data. These new data challenges require us to move from conventional processing to new paradigms to better exploit and optimize the IO system architecture to support these needs (Adrian 2013). The increase in computational bandwidth brings a massive increase in memory bandwidth requirements. High bandwidth memory (HBM) (Jun et al. 2017) enabled FPGA accelerators address these challenges to provide a large memory bandwidth. The use of accelerators as co-processors is the most prevalent way to achieve performance gains to compensate for the slowdown of Moore’s Law. However, there are limiting factors in this approach such as 1) device driver overheads 2) operating system code path length 3) and CPU overhead required in managing the IO requests. Coherent Accelerator Processor Interface (CAPI) was developed to address these challenges, where an accelerator is attached coherently as a peer to the CPU through its IO physical interface (Stuecheli et al. 2015b).
This method significantly increases the performance of the accelerator when compared to the traditional IO model. The performance of a CAPI attached heterogeneous storage device is drastically improved by reducing the overhead of the device driver in the operating system. It also allows direct memory access to the underlying storage without the calls to the device driver and without any intervention from the operating system thereby, reducing the code path length.

The goal is to understand and analyze the performance capabilities of Power8+ CAPI infrastructure through FlashSystem900 (FS900) and compare it with the other heterogeneous memory devices without CAPI. For this experiment, the Key Value (KV) Layer (ArkDB) APIs provided by IBM CAPIFlash library (Code ) was used. The CAPIFlash library provides a multiple of 4K block access to the flash. Our benchmark application uses this facility provided by the library to read a 4K block for the values. The benchmark results directly relate to real-world application performance. The peak performance (IO/s, OP/s and execution time) of the heterogeneous storage devices, storage and computational efficiency are the main focus of this work.

5.2 Background

The main problem observed with traditional flash storage is the CPU overhead leading to stranded IOs (Solution.Ref.Guide ). There was a CPU overhead in managing the IO requests as it flows through the CPU to the storage device. Even with the multi-core architecture, more CPUs are needed to handle the IO request. This results in the wastage of significant portions of the available CPUs that are tied to the IO overhead, rather than doing the useful computations. For applications that require high computational power, multi-core processors often do not suffice to provide high performance. The use of hardware accelerators such as GPUs, ASICs, and FPGAs can provide increased performance for massively parallel programmable architectures for a wide range of applications.

On each Power8+ processor, there is one CAPI interface called Coherent Accelerator Processor
Proxy (CAPP). The PCIe Host Bridge (PHB) on the processor connects to the PCIe IO links. The CAPP unit together with PHB acts as memory coherence, data transfer, interrupt, and address translation agents on the Symmetric Multi-Processor (SMP) interconnect fabric for PCIe-attached accelerator (Starke et al. 2015). This FPGA accelerator has a Power Service Layer (PSL) which provides address translation and system memory cache for the Accelerator Function Unit (AFU) and is connected to the Power8+ processor chip by the PCIe link. The combination of PSL, PCIe link, PHB, and CAPP enhance the capabilities of AFUs. AFUs operate coherently on the data in memory, as peers of other caches in the system.

An application which runs on the processor core has a virtual address space on the memory for performing IO with other devices attached to the processor. In a heterogeneous compute cluster without CAPI, a hardware accelerator such as an FPGA is attached using a PCIe. This leads to having many separate copies of the data, thereby adding overhead to using the FPGA. Since the copies of the data reside in different memory address spaces, any changes that were made to the data by the application would not be coherent with that on the FPGA.

With CAPI FPGA attached to a PCIe, the PSL layer is used to access shared memory regions and cache areas as though they were a processor in the system. This ability enhances the performance of the data access and simplifies the programming effort to use the storage device. Instead of treating the hardware accelerator as an IO device, it is treated as a co-processor, which eliminates the requirement of a device driver to perform communication and the need for Direct Memory Access that requires system calls to the operating system (OS) kernel. By removing these layers, the data transfer operation requires much fewer clock cycles in the processor, improving the IO performance (Caldeira et al. 2015).

Hence with CAPI, there are fewer overheads in IO, thereby minimizing thousands of instructions. All the data is coherently managed by the hardware. Furthermore, an FPGA accelerator can now act as an additional core in the server with a coherent memory. CAPI enables many heterogeneous
memory devices to coherently attach to the Power8+ processors, facilitating an environment to connect various high bandwidth IO devices (CAP a). By using CAPI, terabytes of the flash storage array can be attached to the Power8+ CPU via an FPGA. CAPI enables applications to get direct access to the hardware storage (a large flash array) with reduced IO latency and overhead, thereby increasing the read/write performance compared to the standard IO-attached flash storage (Solution Ref. Guide). CAPI also eliminates the context switch penalties caused by interrupts. All these benefits enable CAPI to provide a hybrid computing environment. More information on the Coherent Accelerator Processor Interface on Power8+ Processor chip are in the CAPI user manual (CAP b).

5.3 CAPIFlash Design

In the Linux kernel, the Coherent Accelerator Interface (CXL) is designed to allow the coherent connection of accelerators (FPGAs and other devices) to a Power processor (Gilge & Orlando 2013). Through LibCXL, the user-space applications can directly communicate to a device (network or storage) bypassing the typical kernel/device driver stack. The CXL flash adapter driver enables direct access to flash storage for a userspace application. Applications which need access to the CXL Flash from the user space should use the CAPIFlash library. More information about the interfacing between CXL and the CAPIFlash is provided in (CAPI.Linux.manual).

The CAPIFlash library IBM Data Engine for NoSQL – Integrated Flash Edition (Code) was built on Power8 systems with CAPI. This library helps to create a new tier of memory by attaching up to 57 terabytes of auxiliary flash memory to the processor. This library provides two sets of public APIs for reading and writing to the physical address space on the flash device: 1. Cflash - Block Layer APIs and 2. ArkDB - Key Value (KV) Layer APIs. Our benchmark is focused mainly on the ArkDB KV layer APIs. The ArkDB KV layer API provides synchronous and asynchronous read/write requests to the flash memory.

The send and receive operations to the KV Store (ArkDB) on the intended device (FS900,
RAM, NVM, SSD) can be either synchronous or asynchronous. In synchronous operation, with a read/write request, the operation is initiated, following which the process is blocked and the system waits for the completion of the process. During this time, the ArkDB threads store and retrieve the data from the KV database instance (ArkDB) on the intended devices through CAPI.

In an asynchronous operation, the processes run in non-blocking mode. It initiates the operation and does not wait for the completion to start the next operation. The caller would discover the completion of the operations later by polling the ArkDB. Since the processes are non-blocking in asynchronous message passing setup, some computations can be performed when the message is in transit, thereby allowing more parallelism.

5.3.1 CAPI Adapter

The PCIe3 LP CORSA CAPI fibre channel Flash Accelerator x8 adapter (FC EJ16; 04CF) (Nallatech 385A72 with Intel Altera) FPGA accelerator card (with two 4GB DDR3), acts as a co-processor for the Power8+ processor. This is designed to offload CPU access to external fiber channel flash storage. The adapter requires a direct-attach, point-to-point 8Gb fiber channel link to external storage, such as IBM FS900. This FPGA accelerator card includes the CAPI PSL, AFU (CAP b) and interfaces to fiber channel IO ports to allow direct memory access to an IBM Flash System (Solution.Ref.Guide ). The EJ1K CAPI flash accelerator leverages the ability to provide high throughput, low latency connection to flash memory to address the scaling problems found in typical flash deployments (Solution.Ref.Guide ). The clock rate of the PCIe bridge is 33 MHz.

5.3.2 IBM FlashSystem 900

FS900 uses FPGA based flash arrays without involving processors and is cost effective when compared to DRAM (Gilge 2013). The flash memory in the IBM FS900 has higher performance due to hardware only data path whereas a traditional SSD based flash memory is typically limited by the
software processing. The IBM FS900 is connected to the CAPI accelerator through a fiber channel. This storage device is configured at RAID 5 and has a usable capacity of 20 TB. It also enables a distributed random-access memory and allows massive data parallelism. FS900 has persistent memory whereas RAM does not. The maximum 4KB IOPS 100% random read is 1,100,000 and 100% random write is 600,000.

We used one CAPI accelerator card, which has two ports, with the World Wide Port Name (WWPN) for each port mapped to a single volume. The KV experiments on the IBM FS900 were performed by setting the two volumes of FS900 (each 10 TB) in superpipe mode. This enables hardware acceleration, with the CPU offloading IO to the device.

5.3.3 NVM and SSD

NVMe is a host controller interface and a storage protocol, designed to accelerate the speed of data transfer using the PCIe bus, through processor-based storage solutions. It can read/write NAND flash memory to deliver the full potential of non-volatile memory in PCIe-based solid-state storage devices. The NVM in Power8+ system uses PCIe3, 3.2 TB NVMe Flash x4 adapter CCIN non-volatile memory controller (HGST Inc Ultrastar SN100) series with Non-Volatile Memory Express (NVMe) SSD. For the PCIe Interconnect, each POWER8 processor has 32 PCIe lanes running at 9.6 Gbps full-duplex. The theoretical bandwidth is: 32 lanes \times 2 processors \times 9.6 \text{ Gbps} \times 2 = 153.6 \text{ GB/s} \ (\text{Caldeira et al. 2015}).

The Samsung SATA SSD (MZ7LM3T8HCJM) used in the S822LC system has a capacity of 3.840 TB with a form factor of 2.5” inches. This SSD use a V-NAND technology and has a data transfer rate of 600 MBps. There are 2 SSDs in this Power8+ system and are connected to the integrated SATA controller in the motherboard.
5.3.4 RAM

RAM in the IBM Minsky system has a total capacity of 1TB. The Power8+ S822LC computing server provides 8 DIMM memory slots each with 128 GB, allowing for a maximum system memory of 1024 GB DDR3 ECC at 1333 MHz. Each Power8+ processor has four memory channels running at 9.6 Gb/s capable of reading 2 Bytes and writing 1 byte at a time. The total theoretical memory bandwidth is: 4 channels * 9.6 Gb/s * 3 Bytes = 115.2 GB/s per processor module (Caldeira et al. 2015).

5.3.5 Processor

This CAPI enabled system S822LC server has two 64-bit IBM Power8+ processors with 10 cores each with 8 threads/core, running at a clock rate of 4 GHz. This system runs Ubuntu OS (version - 16.04.5 LTS ) on the PPC64LE architecture. It has a 64KB D cache, 32KB I cache, 512KB private L2, 8MB L3 per core (96M) on a 22nm chip. The processor to memory bandwidth is 170 GB/s per socket i.e. 340 GB/s per system and IO bandwidth of 64 GB/s simplex. Each Power8+ processor has 2 memory controller. Each memory controller has a two buffer L4 cache and each of them are connected to four RAM DIMM slots (Caldeira et al. 2015). POWER8+ is a revised version of the original 12-core POWER8 from IBM. The main new feature is the support for Nvidia’s bus technology NVLink, connecting up to four NVLink devices directly to the chip. IBM removed the “A Bus and PCI interfaces” for SMP connections to other POWER8 sockets and replaced them with NVLink interfaces.

5.4 Evaluation

5.4.1 Dataset

The input data used for this work is NASA’s MODIS (MODerate-resolution Imaging Spectrometer ) Terra/Aqua Surface Reflectance data. Surface reflectance is the amount of light reflected
by the surface of the earth. In addition to the geo-location coordinates, the data contains several fields collected every 5 minutes at 250m, 500m and 1km resolution as 8-bit, 16-bit or 32-bit signed float/integer types. A sample structure for a refined subset of this dataset is shown in Table 5.1.

The data attributes that are of interest for this study are: Key, Value4, Value5 and Value6 with each entity of size 8 Bytes with a total of 32 Bytes per row in the input data. In our experiments, the dataset ranges from 10 million KV pair records (543 MB) to 100 billion KV pair records (3.1TB).

### 5.4.2 Performance Metrics

Some of the performance metrics used for this experiment are as follows. **OP/s** is the number of Set(write)/Get(read) operations per second submitted by the CAPIFlash library. **IO/s** or IOPS is the number of IO operations per second. **Read time** / **Write time** is the time taken to complete entire reading / writing of data (in seconds).

### 5.4.3 CAPI Flash parameters

The following are the important parameters used for evaluating this benchmark; 1. **Hcount**, 2. **Threads**, 3. **Queue Depth**, 4. **Metadata Cache**.

The CAPIFlash library provides access to the flash in multiples of 4K blocks. The **hcount** (-h) parameter is a calculation of how the KV pairs fit into a physical 4K block (Code ). This, in turn, provides a more balanced allocation of storage space, thereby increasing the performance and storage efficiency. **Hcount** is calculated by dividing 4K by (32 (Bytes per KV entry in the dataset) +

---

**Table 5.1. Data fields in the MODIS Surface Reflectance dataset**

<table>
<thead>
<tr>
<th>No:</th>
<th>Latitude</th>
<th>Longitude</th>
<th>Day</th>
<th>Band 1</th>
<th>Band 2</th>
<th>Band 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80.647079</td>
<td>34.720413</td>
<td>67</td>
<td>-28672</td>
<td>-28672</td>
<td>-28672</td>
</tr>
<tr>
<td>2</td>
<td>80.658272</td>
<td>34.457428</td>
<td>67</td>
<td>-28672</td>
<td>-28672</td>
<td>-28672</td>
</tr>
<tr>
<td></td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
8 Bytes (metadata)). Hence, a value of 100 would give the best performance for a dataset where each record is of 32 Bytes. The value 100 in this calculation means that the read/write operations would be performed on 100 KV pairs in parallel in a given 4K block. The value of hcount is calculated as the total number of KV pairs in the input data divided by 100 in this example. The total memory allocated in Bytes (inuse) would then be close to the Bytes written to the KV instance (actual), thus providing an optimal storage efficiency.

There are two different threads with regards to this benchmark: 1. application threads and 2. ArkDB threads. Each read / write is an operation submitted by application thread(s) to the ArkDB, and then the application threads poll the ArkDB for their completion. There is a maximum number of operations that can be queued to the ArkDB. The maximum value for Queue Depth (QD) is 1024 with a default of 400. The queued operations are then submitted by the ArkDB threads to the storage media. QD represents the number of ArkDB operations which may run in parallel.

In synchronous mode, an application thread sends a command (read/write) and waits for completion while the ArkDB threads do the work. As a result, one thread can do QD=1. More threads are required to handle a greater value of QD. In asynchronous mode, each operation submitted adds to the QD. A certain QD is required to reach the maximum bandwidth of the card. In asynchronous mode, with more threads, more CPU is used. The threads can be increased until the maximum performance is obtained. In asynchronous mode, one thread may submit any number of operations (QD=N) and then check for the completion of each command. The ArkDB threads do the database side of the work. Too many ArkDB threads increase CPU usage without increasing OP/s and reduce the overall performance. Hence finding an optimal thread for the peak performance is important. The threads mentioned in the remainder of the experiment are the ArkDB threads. The number of threads can be passed in as an input parameter which specifies the total ArkDB threads.

FS900 can be used either with MetaDataCache turned on or off. This allows a part of the RAM to be used as a cache for the data access to the FS900. Note that, the number of IO/s for FS900
which are greater than 400K are from the ArkDB cache hits, which is RAM. In order to get the
maximum performance for the benchmark, all FS900 experiments were run with ArkDB metadata
cache hits (RAM) enabled unless specified.

5.4.4 Benchmark Application

The benchmark application used, runs a simple read and write operation to the KV store
(ArkDB) on the intended devices (FS900, RAM, NVM, SSD) using the KV layer APIs of the
CAPIFlash library. Only FS900 uses the CAPI accelerator card in the real IO mode, whereas
NVM and SSD do not use the CAPI accelerator and are configured in the file IO mode. When the
benchmark application requests to run to a file on NVM or SSD, the Block API diverts to the file
before reaching the CAPI card (Solution.Ref.Guide). When the benchmark application requests to
run on RAM, then the ArkDB API diverts the calls to RAM, before reaching the CAPI card. The
CAPI card is not used when running the application to RAM. FS900 experiments were run with
enabling the metadata cache (RAM). When an ArkDB operation gets a hit on the ArkDB metadata
cache, the call does not go to the CAPI card or FS900. When the maximum bandwidth of a single
Corsa CAPI card reaches 380K IO/s, any OP/s higher than 380K, are due to the ArkDB cache
hits. We ran the KV R/W experiments by varying the threads from 1 to 128 and QD = 400 in both
synchronous and asynchronous modes to find out the peak performance of each device. Finally, we
summarize the overall read/write performance for all storage devices. The CAPIFlash library version
used in this experiment is 5.0.2706.

5.4.5 Performance of Write Operations

The results of Write performance of FS900, RAM, NVM and SSD for a dataset of 10 million KV
pairs (0.5 GB) are in Figure 5.1. In synchronous Write IO/s graph, for FS900, NVM and SSD, there
was a plateau in the graph between 0.4 million to 0.8 million IO/s after 10 threads, with FS900 giving
the maximum IO/s out of the three devices. For RAM, after 40 threads, the IO/s plateaued around 3.25 million after 40 threads. A similar trend was observed in the synchronous OP/s graph. With asynchronous Write IO/s graph, increasing the number of threads increased the IO/s performance in RAM until 5 million IO/s. There was little variation in the case of FS900, SSD and NVM. A similar trend was observed in the asynchronous OP/s graph. In the case of analyzing the time to completion for read/write operations, having a lower value is better. For write operations, the optimal threads for all devices were found around 30 threads. After 30 threads, each of the devices plateaued around a band of certain time regions without decreasing much time. From figure 5.1 for write operations, it is clear that the best performance was found in the following order: RAM, FS900, NVM and SSD. Asynchronous mode was able to give the peak performance. RAM consumed more numbers of threads to reach peak IO/s and OP/s, whereas for FS900, 10 threads were enough to reach the peak performance and lowest completion time. The performance in asynchronous mode was found to be better since the computations do not block the process to wait for its completion, as opposed to synchronous mode. In asynchronous mode, the ArkDB threads continue the computations while the messages are in transit and this approach allows more parallelism. The main advantage of FS900 with CAPI is the use of lesser CPU resources (if CPU threads are crucial to the application). Throughout the write operation, it is observed that using too many ArkDB threads decreases performance without increasing IO/s or OP/s and reduces the latency. For write operations, SSD and NVM showed similar performances as FS900.

### 5.4.6 Performance of Read Operations

Figure 5.2 depicts the performance comparison on all the devices for reading 10 million KV pairs. For synchronous mode, the read IO/s for all devices lie around 1.5 million to 4.5 million, whereas in asynchronous mode, increase in threads almost gave a linear increase in read IO/s. In synchronous mode, FS900, NVM and SSD gave the maximum read IO/s compared to RAM. In the
Fig. 5.1. Write performance for FS900, RAM, NVM and SSD
Fig. 5.2. Read performance for FS900, RAM, NVM and SSD
case of asynchronous, FS900 gave the maximum performance of 17 million IO/s. SSD and NVM at 10 million IO/s and RAM at 6 million IO/s. For OP/s, FS900 performed better than RAM in both synchronous and asynchronous modes. FS900 took less time to complete the read operation in comparison with RAM. This was possible even with lesser number of threads. Based on our analysis, RAM is well suited for write-intensive workloads. For read operations, FS900 was found to be better than RAM. Having a RAM of large capacity comes at a higher overall cost. Similar or near to RAM performance could be achieved with FS900 at a lesser cost (Gilge 2013). CAPI facilitates attaching terabytes of the flash storage array to the Power8+ CPU with reduced IO latency and overhead compared to a standard IO-attached flash storage. FS900 with CAPI, when using the RAM metadata cache, performed 2x as many read operations in synchronous mode and 3x in asynchronous mode. The lower performance of RAM may be due to the CPU overhead required in managing IO operations. Without metadata cache, FS900 gave a steady 380K IO/s in both the modes consistently for all threads above 10.

5.4.7 Summary of Read Write performance of RAM, FS900, NVM and SSD

The peak R/W performance for 10 million KV pairs on all the devices is summarized in Figure 5.3. The best values of read and write OP/s, IO/s and time are compared for all the devices. RAM outperformed all the other devices for write in terms of OP/s, IO/s and completion time. Hence for write-intensive applications, RAM would provide the best performance. Although RAM performed the best for write operations, FS900 provided the highest IO/s for reads. To read 10 million KV pairs, FS900 had the peak IO/s and lowest read completion time. RAM was 2x slower than the other devices in asynchronous mode and 1.3x slower in synchronous mode. After 128 threads, the performance plateaued.

Based on the above analysis, we chose an optimal thread (40), QD (400) and compared the performance of synchronous and asynchronous modes on all devices for 30 million KV pairs (1.6
GB dataset) and the results are shown in Figure 5.4 and 5.5. Here, we also show the difference in the performance between FS900 with meta data cache (WMC) enabled and without meta data cache (WoMC). For all the devices, asynchronous mode gave the peak performance and the lowest completion time, with FS900 giving the peak read IO/s compared to other devices.

Figure 5.6 & 5.7 details the CPU utilization (% CPU and SYS % CPU) for 30 million KV pairs. The graphs start with a straight line indicating the Write operation. The change in the slope of the line indicates that the write operation is completed and the Read operation has begun. This second part of the line indicates the CPU consumption during read operation. The series lines end at different times for different devices, indicating the faster completion time. With regards to % CPU consumption, FS900 in asynchronous is found to be an efficient choice next only to RAM. Considering
Fig. 5.4. IO/s performance of FS900, RAM, NVM and SSD for 30 Million KV dataset

Fig. 5.5. Read Time and Write Time of FS900, RAM, NVM and SSD for 30 Million KV dataset
Fig. 5.6. CPU% used by the application process for 30 million KV dataset

Fig. 5.7. CPU% used by the system process for 30 million KV dataset
the fact that FS900 gives the peak performance with IO/s and OP/s, this minimal difference in CPU consumption can be eliminated. With % SYS CPU consumption, the time spent on running kernel space system process, FS900 asynchronous with metadata cache enabled is found to be an optimal choice, especially for reads. The reason for the limited CPU consumption is due to the fact that, the total number of instructions per IO is reduced with CAPI. This leads to significant improvement in processor time spent in managing the IO. This, in effect, frees up processing resources for actual compute work and not just moving around data (Solution.Ref.Guide).

### 5.4.8 Performance of ark\_nextN API

A normal read (without NextN) calls ark\_get API for each key in the ArkDB to return the value for the key, and these operations are done in parallel based on QD. Whereas, the ark\_nextN API returns each key and not its value in the ArkDB, in a pseudo-random order, one operation at a time. The ark\_next API reads one KV pair at a time for each 4K reads. ark\_nextN API gets “N” keys at a time for each 4K reads in the ArkDB store. For the dataset in this study: key=8 Bytes and values =24 Bytes. This is about 100x faster for just reading the keys.

<table>
<thead>
<tr>
<th>Device</th>
<th>G Value</th>
<th>OP/s</th>
<th>IO/s</th>
<th>Time, (seconds)</th>
<th>OP/s</th>
<th>IO/s</th>
<th>Time, (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSD</td>
<td>1</td>
<td>39,840</td>
<td>326,236</td>
<td>251</td>
<td>6,086</td>
<td>48,861</td>
<td>1.643</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>40,000</td>
<td>356,924</td>
<td>250</td>
<td>5,250</td>
<td>42,148</td>
<td>2</td>
</tr>
<tr>
<td>NVM</td>
<td>1</td>
<td>41,152</td>
<td>367,205</td>
<td>243</td>
<td>6,644</td>
<td>53,341</td>
<td>1.505</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>41,152</td>
<td>367,205</td>
<td>243</td>
<td>5,250</td>
<td>42,148</td>
<td>2</td>
</tr>
<tr>
<td>FS900</td>
<td>1</td>
<td>285,714</td>
<td>727,040</td>
<td>35</td>
<td>4,411</td>
<td>7,832</td>
<td>2.267</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>294,117</td>
<td>748,424</td>
<td>34</td>
<td>3,973</td>
<td>5,772</td>
<td>23</td>
</tr>
<tr>
<td>RAM</td>
<td>1</td>
<td>588,235</td>
<td>1,496,848</td>
<td>17</td>
<td>86,956</td>
<td>154,396</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>555,555</td>
<td>1,413,698</td>
<td>18</td>
<td>91,385</td>
<td>132,769</td>
<td>1</td>
</tr>
</tbody>
</table>
With $G=1$ vs $G=1000$, there was around 800x improvement on read time from SSD and NVM and 100x improvement for FS900 and RAM. Table 5.2 also indicates that, even though the read IO/s and OP/s was not the maximum or the optimal for the device, there was a significant reduction in the execution time with the use of \texttt{ark.nextN} API. This was also the reason why the peak IO/s and OP/s was the focus of this study rather than reducing the execution time.

### 5.4.9 Large Experiments Results

Using a CAPI Interface, FS900 performs better for read operations than the other devices, including RAM. This becomes highly advantageous when the dataset size is very large. Table 5.3 shows the results of the performance experiments on larger datasets (120GB, 954GB, 1500GB and 3TB). The OP/s and IO/s for FS900 were almost consistent, irrespective of the dataset size. In the case of RAM, the performance of read/write OP/s and IO/s increased by around 1.5 times until the dataset size is less than the size of RAM (1 TB). When the dataset size gets larger than the size of RAM, RAM can have cache misses and the virtual memory manager would be using the system disk for swapping, thereby increasing the paging and latency. At this stage, we observed a drastic decrease in the performance of IO/s and OP/s for RAM. For the dataset with 100 billion KV pairs (3.17 TB), FS900 provided a steady 380K IO/s for 71 hours.

<table>
<thead>
<tr>
<th>Device</th>
<th>Dataset Size</th>
<th>Write OP/s</th>
<th>Write IO/s</th>
<th>Time (seconds)</th>
<th>Read OP/s</th>
<th>Read IO/s</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS900</td>
<td>120 GB</td>
<td>166,223</td>
<td>331,259</td>
<td>18,048</td>
<td>358,808</td>
<td>381,680</td>
<td>8,361</td>
</tr>
<tr>
<td></td>
<td>954 GB</td>
<td>179,750</td>
<td>357,695</td>
<td>138,386</td>
<td>383,222</td>
<td>383,222</td>
<td>64,910</td>
</tr>
<tr>
<td></td>
<td>1.5 TB</td>
<td>188,628</td>
<td>375,354</td>
<td>210,148</td>
<td>383,822</td>
<td>383,822</td>
<td>103,277</td>
</tr>
<tr>
<td></td>
<td>3.17 TB</td>
<td>189,564</td>
<td>377,477</td>
<td>515,391</td>
<td>382,732</td>
<td>382,732</td>
<td>255,270</td>
</tr>
<tr>
<td>RAM</td>
<td>120 GB</td>
<td>2,941,176</td>
<td>5,861,339</td>
<td>1,020</td>
<td>4,716,981</td>
<td>5,017,660</td>
<td>636</td>
</tr>
<tr>
<td></td>
<td>954 GB</td>
<td>3,539,413</td>
<td>7,043,255</td>
<td>7,028</td>
<td>5,327,693</td>
<td>5,327,693</td>
<td>4,669</td>
</tr>
<tr>
<td></td>
<td>1.5 TB</td>
<td>2,149,267</td>
<td>4,283,774</td>
<td>16,936</td>
<td>2,576,991</td>
<td>2,576,991</td>
<td>14,125</td>
</tr>
</tbody>
</table>
In general, RAM is faster for most of the operations, but it comes with a limited capacity (1TB). FS900 has 20TB of storage and can also persist the data. The effectiveness of FS900 with CAPI comes into picture only when the dataset size is very large compared to the size of RAM. The CAPI 2.0 in Power9 makes use of the PCIe Gen4 and provides double the performance.

5.5 Related Works

There are various studies on benchmarking and evaluating heterogeneous storage systems. Apache Cassandra Optimized durable commit log using CAPI-Flash (Sendir et al. 2016) focuses on durable logging on flash using Power8 CAPI-Flash. Their CAPIFlash commit log showed a 107% better throughput in write-only workloads compared to Cassandra’s durable alternative.

Our work is the first to study the performance of the FS900 with CAPI accelerator card through CAPIFlash read write benchmark. We also evaluated the performance by varying the threads and other parameters of the CAPIFlash library and compared it with RAM, SSD and NVM without using the CAPI accelerator. We evaluated the read time / write time, OP/s and IO/s of RAM, SSD, NVM and FS900 with both synchronous and asynchronous modes. We also found the optimal CPU threads and other parameters of the CAPIFlash library to attain a peak performance. These results for different memory devices, and their communication modes when using CAPIFlash library, will be helpful for developers in estimating their applications design choices.

5.6 Conclusions

The performance of the FS900 with CAPI accelerator card was evaluated through a basic read/write benchmark and compared it to various heterogeneous storage devices. The asynchronous mode on all devices gave the best results with IBM FS900, giving the highest read IO/s. The peak performance on FS900 with CAPI was observed using a lesser number of threads. This enables the CPU resources to be efficiently used in other areas of the application. When the dataset size exceeds
the capacity of RAM, FS900 is a cost-effective alternative.

The flash memory in the IBM FS900 has higher performance due to hardware only data path, its distributed random-access memory and its allowance of massive parallelism in handling the data. The CAPI accelerator allows access to shared memory regions and cache areas as though they were a processor in the system. The CPU overhead in managing the IO request is now offloaded to the CAPI accelerator. The data transfer operation minimizes thousands of instructions and requires much fewer clock cycles in the processor. This improves the read/write performance of FS900 with CAPI unlike standard IO-attached flash storage. CAPI enables the class of IO-bound applications to perform better, where these advantages are a critical factor.
CONCLUSION AND FUTURE WORK

This thesis explores the effectiveness of data-centric compute architectures using Active Storage, Processing In-Memory and Coherent Access Processor Interface (CAPI) accelerated Flash storage with the aim to reduce the data movement between storage and compute units and between processor units and memory. We developed a compute framework Active In-Storage (AiSTOR) for Scalable Distributed Big Data Processing on active storage devices to perform the computations directly in the storage devices. AiSTOR has the following advantages: First, active storage utilizes the processing power of storage devices and this significantly reduces the bandwidth requirement on the network. Second, the computations on the storage devices can take advantage of storage parallelism by using an array of active storage devices, which aggregate the processing power of many devices. Third, it can address the coherent processing streaming data as it arrives for storage. In comparison with Hadoop based MapReduce the compute times on AiSTOR has significant performance benefits by up to 18%, while providing very competitive results compared to Spark-based in-memory processing.

We define a generic near data processing architecture, well-suited for memory-bound computations, and implement software kernels for NDP-based algorithmic mapping. We show that the algorithms for a full-system NDP architecture framework yields improved accurate performance. The effectiveness of the NDP architecture is demonstrated by evaluating the row-buffer management policies (open-page and closed-page) with controller modifications and a generic unmodified archi-
tecture. The PIM open-page policy has 52% higher operation throughput than the host and 3.7% higher throughput and 50% lesser DRAM activations than PIM-closed page policy. Further, this dissertation explores the impact of CPU usage in handling the IO requests in heterogeneous storage systems when using CAPIFlash library and finding the optimal IO/s and OP/s for heterogeneous storage/memory devices such as NVM, SSD and RAM. FS900 with CAPI, when using the RAM metadata cache, performed 2x as many read OP/s in synchronous mode and 3x in asynchronous mode. SSD and NVM had 66% higher IO/s in comparison with RAM in asynchronous mode.

6.1 Summary of Contributions

This work makes several contributions to the understanding and analysis of how to efficiently use Near Data Processing for mitigating the inefficiencies of compute-centric architectures by taking advantage of the computational power of in-storage processing devices, near-memory processors and coherent processing software systems.

AiSTOR: an Active In-Storage compute framework for program execution within the ISP devices

One of the main contributions of this work is the development of Aistor, an Active In-Storage compute framework for efficient Big Data processing. Aistor provides transparent and multi-granularity processing for program execution within the ISP devices. Aistor is based on a stream based programming model which provides a simple and convenient way for users to apply several application functions on the data object, a set of commonly used functions are included in an applet. This facilitates ease of programming since the user can easily access this functionality by passing the function name and the required attributes. It also enables chaining multiple functions on a single object. AiSTOR performs runtime resource monitoring by loading an executable to fetch the applet status while it is running on the storage device. The runtime resource monitoring metrics such as CPU utilization on the storage device and the temperature of the processor cores could be used for load balancing.
PIM: Near Memory Acceleration of Memory-bound algorithms This thesis defines a generic near data processing architecture which is well-suited for memory-bound computations and implement the software kernels for NDP-based mapping for the algorithms on a cycle accurate full-system NDP architecture framework which yields a realistic detailed performance analysis. Evaluate the row- buffer management policies (open-page and closed-page) on an NDP architecture with the controller modifications and a generic unmodified architecture. Further, this study provides a characterization of the NDP-mapped memory-bound algorithms and demonstrates the shortcomings of other implementations.

CAPI: Coherent Accelerator Processor Interface for Heterogeneous memory/storage devices This work analyzes the impact of CPU usage in handling the IO requests in heterogeneous storage systems when using CAPIFlash library and finding the optimal IO/s and OP/s for heterogeneous storage devices such as NVM, SSD and RAM. For applications that require high computational power, multi- core processors often do not suffice to provide high performance. The use of hardware accelerators such as GPUs, ASICs, and FPGAs can provide increased performance for massively parallel programmable architectures for a wide range of applications. The performance of CAPI accelerator was evaluated by varying the threads and other parameters of the CAPIFlash library and compared it with RAM, SSD and NVM without using the CAPI accelerator. This work is the first to study the performance of the FS900 with CAPI accelerator card through CAPIFlash read write benchmark.

6.2 Future Work

Based on the analysis of the many existing NDP architectures, the following areas of future research could further explore processing close to memory/storage.

- Processing near heterogeneous memories is a new research area with high potential for enhancing applications in the area of AI and machine learning. As hybrid designs tightly integrate CPU,
DRAM, and a flash-based storage such as NVM, the PIM approach could be explored to meet
the needs application processing large amounts of data i.e., with larger capacity, smaller delay,
and wider bandwidth. Further the research on heterogeneous memories could be extended to
study the novel high-density byte-addressable memory such as Intel Optane DC Persistent
Memory (Optane PMM) (Gill et al. 2019) which is a cost-effective alternative to DRAM with
faster data access times than SSDs.

- More quantitative exploration in the area of interconnect networks between the memory/storage
and compute units is required. NDP units with emerging interconnect standards such as GenZ
(M.Krause.and.M.Witkowski ), CXL (Sharma 2019), CCIX (Consortium & others ) and
OpenCAPI (Stuecheli et al. 2018) could be vital in improving the performance and energy
efficiency of big data workloads running on NDP enabled compute systems. CXL features a
CPU-to-device interconnect that targets high-performance workloads on heterogeneous compute
engines. The coherency and low latency of CXL are enabled by dynamically multiplexed cache
and memory (K.Rao 2020). The CXL cache allows an accelerator to cache system memory
while the CXL memory gives the host processor access to the memory which is attached to the
accelerator. The accelerator memory could be attached to the coherent space of CPU and this
expands its available address space (J.Russel 2020).

- The field of NDP requires a generic set of tools for the novel architectures for application
characterization and offloading processing and data to NDP systems. This requires new metrics
to access the application performance which could aid in deciding whether an application that
should be offloaded to an NDP system.

As scientific research and applications continue to generate large amounts of data, the principles
proposed in this thesis are expected to reach main stream products, and this would prompt further
research in the near future.
REFERENCES


[29]

[30]


Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. NSDI.


Conference on Advances in Computing, Communications and Informatics (ICACCI), 1458–1464. IEEE.


[147] Reinsel, D.; Gantz, J.; and Rydning, J. 2020. The digitization of the world from edge to


[201] Zhao, J.; Feng, X.; Cui, H.; Yan, Y.; Xue, J.; and Yang, W. 2013. An empirical model for predicting cross-core performance interference on multicore processors. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, 201–212. IEEE.
