UNIVERSITY OF CALIFORNIA

Los Angeles

Architectural Techniques to Enhance the Efficiency of Accelerator-Centric Architectures

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

Yuchen Hao

2018

© Copyright by
Yuchen Hao
2018

ABSTRACT OF THE DISSERTATION

Architectural Techniques to Enhance
the Efficiency of Accelerator-Centric Architectures

by

Yuchen Hao
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2018
Professor Glenn D. Reinman, Chair

In light of the failure of Dennard scaling and recent slowdown of Moore's Law, both industry and academia seek drastic measures to sustain the scalability of computing in order to meet the ever-growing demands. Customized hardware accelerator in the form of specialized datapath and memory management has gained popularity for its promise of orders-of-magnitude performance and energy gains compared to general-purpose cores. The computer architecture community has proposed many heterogeneous systems that integrate a rich set of customized accelerators onto the same die. While such architectures promise tremendous performance/watt targets, our ability to reap the benefit of hardware acceleration is limited by the efficiency of the integration.

This dissertation presents a series of architectural techniques to enhance the efficiency of accelerator-centric architectures. Staring with physical integration, we propose the Hybrid network with Predictive Reservation (HPR) to reduce data movement overhead on the on-chip interconnection network. The proposed hybrid-switching approach prioritizes accelerator traffic using circuit-switching while minimizes the interference caused to regular traffic. Moreover, to enhance the logical integration of customized accelerators, this dissertation presents an efficient address translation support for accelerator-centric architectures. We observe that accelerators exhibit page split phenomenon due to data tiling and immense

sensitivity to address translation latency. We use this observation to design two-level TLBs and host page walk to reduce TLB misses and page walk latency, which provides within 6.4% of ideal performance. Finally, on-chip accelerators are only part of the entire system. To eliminate data movement across chip boundaries, we present the compute hierarchy which integrates accelerators to each level of the conventional memory hierarchy, offering distinct compute and memory capabilities. We propose a global accelerators manager to coordinate between accelerators in different levels and demonstrate its effectiveness by deploying a content-based image retrieval system.

The techniques described in this dissertation demonstrate some initial steps towards efficient accelerator-centric architectures. We hope that this work, and other research in the area, will address many issues of integrating customized accelerators, unlocking end-to-end system performance and energy efficiency and opening up new opportunities for efficient architecture design.

The dissertation of Yuchen Hao is approved.

Jingsheng Jason Cong

Milos D. Ercegovac

Anthony John Nowatzki

Glenn D. Reinman, Committee Chair

University of California, Los Angeles

2018

Dedicated to my family, friends and mentors.

TABLE OF CONTENTS

1 Introduction 1
1.1 Focus of This Dissertation: Accelerator-Centric Architectures 2
1.1.1 Key Challenges for Integrating Customized Accelerators 2
1.1.2 Goal of This Dissertation 3
2 Background and Related Work 5
2.1 Accelerator-Centric Architectures 5
2.1.1 Loosely-Coupled Accelerators 5
2.1.2 Near-Memory Accelerators 8
2.1.3 Near-Storage Accelerators 8
2.1.4 Address Translation Support 9
2.2 Simulation Infrastructure 11
3 On-chip Interconnection Network for Accelerator-Rich Architectures 13
3.1 Overview of the Baseline Architecture 15
3.2 Network Characterization 17
3.3 HPR: Hybrid Network and Predictive Reservation 20
3.3.1 Global Management of Circuit-Switching 20
3.3.2 Hybrid Network 22
3.4 Evaluation Methodology 24
3.5 Experimental Results 26
3.6 Summary 29
4 Supporting Address Translation for Accelerator-Centric Architectures 30

4.1 Characterization of Customized Accelerators 34
4.1.1 Accelerator-Centric Architectures 34
4.1.2 Bulk Transfer of Consecutive Data 35
4.1.3 Impact of Data Tiling 37
4.1.4 Address Translation Latency Sensitivity 38
4.2 Evaluation Methodology 40
4.3 Design and Evaluation of Address Translation Support 42
4.3.1 Overview of the Baseline IOMMU Approach 42
4.3.2 Private TLBs 43
4.3.3 A Shared Level-Two TLB 46
4.3.4 Host Page Walks 50
4.3.5 Summary: Two-level TLBs and Host Page Walks 53
4.4 Discussion 55
4.4.1 Impact of More Accelerators 55
4.4.2 Large Pages 57
4.5 Summary and Future Work 58

Appendices for Chapter 4

4.A A Performance Analysis of Modern CPU-FPGA Platforms with a Unified Address Space 60
4.A.1 Overview of Modern CPU-FPGA Platforms 61
4.A.2 Experimental Methodology 63
4.A.3 Effective Bandwidth 64
4.A.4 Effective Latency 67

5 Compute Hierarchy: A Case for Near Data Processing 70

5.1 Compute Hierarchy 71
5.1.1 Accelerator Design 72
5.1.2 Global Accelerator Manager 75
5.2 Case Study: Content-based Image Retrieval 75
5.2.1 Application Overview 76
5.2.2 Application Mapping 78
5.3 Experimental Setup 81
5.4 Experimental Results 82
5.4.1 Overall Performance 82
5.4.2 CBIR Pipeline Performance Breakdown 83
5.5 Discussion 85
6 Conclusion 88
References 90

LIST OF FIGURES

2.1 Overview of the Accelerator-Rich Architecture [CFG15] 6

2.2 The programming model and underlying hardware execution mechanism [CFG15] 6

2.3 Overview of the PARADE simulator [CFG15] 11

3.1 The overview of the accelerator-rich architecture 16

3.2 The microarchitecture of the on-chip accelerator 17

3.3 Network Traffic Breakdown. Results are collected on the baseline architecture described in Section 3.1. Details about the benchmarks and setups of the simulation platform can be found in Section 3.5. 18

3.4 L2 cache bank access trace from the read phase of `Deblur` and `BlackScholes`. A read to a cache bank is depicted as a dot in the figure. 19

3.5 The service timeline of one accelerator execution 21

3.6 The hybrid-switched router architecture 23

3.7 The packet-switched and circuit-switched pipeline 23

3.8 An example of the reservation tables from two neighboring routers 24

3.9 Normalized execution time 26

3.10 Normalized network latency 26

3.11 Normalized energy 27

3.12 Fraction of flits covered by circuit-switched paths 28

3.13 Network latency of control messages 28

4.1 Problems in current address translation support for accelerator-centric architectures in an IOMMU-only configuration 31

4.2 Performance of the baseline IOMMU approach relative to ideal address translation 32

4.3 A detailed look at an accelerator connecting with system IOMMU (baseline) . . 34

4.4 TLB miss behavior of BlackScholes 36

4.5 TLB miss trace of a single execution from BlackScholes 37

4.6 Rectangular tiling on a 32 32 32 data array into 16 16 16 tiles. Each tile accesses 16 pages and can be mapped to a different accelerator for parallel processing. 38

4.7 Geometric mean slowdown over all benchmarks with varied address translation latencies, with LPCIP being the most sensitive benchmark 40

4.8 Performance for benchmarks other than medical imaging with various private TLB sizes, assuming fixed access latency. 44

4.9 Performance for medical imaging benchmarks with various private TLB sizes, assuming fixed access latency. 45

4.10 The structure of two-level TLBs 46

4.11 Page walk reduction compared to the IOMMU baseline23[2] 48

4.12 Page walk reduction from adding a 512-entry shared TLB to infinite-sized private TLBs 49

4.13 Average page walk latency when offloading page walks to the host core MMU . 53

4.14 Average translation latency of (a) all requests; (b) requests that actually trigger page walks 54

4.15 Total execution time normalized to ideal address translation 55

4.16 The average number of outstanding shared TLB misses of the 4-instance and 16-instance cases 56

4.17 Performance of launching 16 accelerator instances relative to ideal address translation 57

4.18 A tale of four CPU-FPGA platforms 62

4.19 Alpha Data PCIe-DMA bandwidth breakdown 64

4.20 Effective bandwidth of Alpha Data, CAPI, HARPv1 and HARPv2 65

5.1 The throughput gap between the host PCIe bandwidth and the distributed SSD bandwidth. This gap grows as we scale-up the SSD units in the system. 71

5.2 The accelerator-interposed memory architecture with the AIMbus which enables inter-DIMM communication. 72

5.3 We distribute an FPGA accelerator to each SSD unit in order to match the distributed bandwidth and utilize the large aggregated bandwidth. 74

5.4 The general pipeline of billion-scale CBIR systems. In this work, we focus on components in the on-line stage – feature extraction, short-list retrieval, rerank and reverse lookup. 76

5.5 Overview of the deployment of CBIR pipeline on the compute hierarchy 80

5.6 Runtime comparison between the PCIe-based acceleration and the compute hierarchy. Each total runtime is stacked by time spent on feature extraction, shot-list retrieval and rerank, from bottom to top, respectively. 83

5.7 Runtime comparison of implementing the feature extraction step using PCIe-based, on-chip, near-memory or near-storage accelerators. 84

5.8 Runtime comparison of implementing the short-list retrieval step using PCIe-based, on-chip, near-memory or near-storage accelerators. 84

5.9 Runtime comparison of implementing the rerank step using PCIe-based, on-chip, near-memory or near-storage accelerators. 85

5.10 Execution time comparison of the three different algorithms deployed. From left to right is the compute hierarchy with exact vectors, compressed vectors and the linear search approach that skips the short-list retrieval step. The y-axis is in log scale. 86

5.11 Execution time breakdown of the three different algorithms deployed. 87

LIST OF TABLES

3.1 System parameters for the baseline architecture 25

4.1 Benchmark descriptions with input sizes and number of heterogeneous accelerators. [CGG14b] 39

4.2 Parameters of the baseline architecture 41

4.3 Configuration of our proposed address translation support 54

4.4 Classification of modern CPU-FPGA platforms 61

4.5 Platform configurations of Alpha Data, CAPI, HARP and HARP2 64

4.6 CPU-FPGA access latency in HARPv1 67

4.7 Hit latency breakdown in HARPv1 68

5.1 An overview of the memory requirement and the compute requirement of each CBIR pipeline stage. 79

5.2 Experimental setup of the compute hierarchy system 81

ACKNOWLEDGMENTS

Many people and organizations have contributed to this dissertation, intellectually, motivationally, or otherwise financially. This is my attempt to acknowledge their contributions.

First of all, I thank my advisor, Glenn Reinman, for providing me with the freedom and resources to do research, for teaching me how to do high-quality research and effectively communicate my ideas to others, and for always being available to chat when I need advise.

I would like to thank Jason Cong, Milos Ercegovac and Tony Nowatzki for serving on my dissertation committee. Special thanks to Jason for his valuable comments and suggestions on my research activities and directions.

Many people have contributed to this dissertation and my life in the past five years. I especially thank:

Peng Wei, Cody Yu Hao, Young-kyu Choi, Jie Wang and Peipei Zhou, for our memorable trips and delicious food, and for their hearty friendship.

Yu-ting Chen and Di Wu for their advises in my early graduate career.

Zhenman Fang, for our collaborations and many technical discussions.

Nazanin Faraphour, for being a very good and cheerful labmate.

Michael Gill, for his inspirations and his contributions to our group's simulation infrastructure.

Tingyang Zhang and Yi Ding, for being very good and caring roommates.

Yunfan Tang, for spending countless hours with me on the game we both love as much as our research.

Many people outside the lab have provided invaluable feedback on my research and helped shape my career. I especially thank my undergraduate advisor, Yu Wang for keeping in touch with me throughout my PhD and always having faith in me. I would like to thank Davy

Huang for being a great mentor and for teaching me important skills in the workplace. I also want to thank Aleksey Pershin for inspiring me to pursue my career in the industry as a persevering and tenacious PhD.

This work is partially supported by the Center for Domain-Specific Computing under the NSF InTrans Award CCF-1436827; funding from CDSC industrial partners including Baidu, Fujitsu Labs, Google, Huawei, Intel, IBM Research Almaden and Mentor Graphics; and C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. Their financial support was vital to my graduate career as it allowed me to remain focused on my research.

Last but not least, I thank my parents for their unconditional love and support throughout my life, even when we are continents away. I owe all my accomplishments to them.

VITA

2011 – 2013	Undergraduate Researcher, Nanoscale Integrated Circuits and Systems Lab, Tsinghua University
2012	Undergraduate Research Intern, Safari Group, Carnegie Mellon University
2012	Intern, Microsoft Asia-Pacific R&D Group, Beijing, China
2013	B.S. (Electrical Engineering), Tsinghua University
2013 – 2014	Department Fellowship, Computer Science Department, University of California, Los Angeles
2013 – 2018	Graduate Researcher, Center for Domain Specific Computing (CDSC), University of California, Los Angeles
2015	Hardware Engineer Intern, Baidu USA, Sunnyvale, CA
2016	Hardware Engineer Intern, Samsung Semiconductor, San Jose, CA
2017	Software Engineer Intern, Facebook, Menlo Park, CA
2017 – 2018	Teaching Assistant, Computer Science Department, University of California, Los Angeles

PUBLICATIONS

J. Cong, Z. Fang, Y. Hao, G. Reinman, "Supporting Address Translation for Accelerator-Centric Architectures", *High Performance Computer Architecture (HPCA)*, 2017.

Y. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, P. Wei, "A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms", *Design Automation Conference (DAC)*, 2016.

J. Cong, M. Gill, Y. Hao, G. Reinman, B. Yuan, "On-chip Interconnection Network for Accelerator-Rich Architectures", *Design Automation Conference (DAC)*, 2015.

CHAPTER 1

Introduction

The semiconductor industry has been driven by Moore's Law and Dennard scaling for the past 40 years. The two scaling laws combined provide us with cost scaling and performance scaling at the same time. As a consequence, the performance of microprocessors has increased by 10,000x, a rate that is faster than any other things our society has ever created [Hen17]. In the 2000s, as microprocessors started to reach their power limit, multicore is our answer to the fading of Dennard scaling. However, transistor scaling does not fundamentally extend Dennard scaling. First, putting more transistors than power budget permits leads to limited utilization of transistors, which is known as dark silicon [EBS11]. Second, Amdahl's Law [Amd67] illustrates that the overall speedup is heavily limited by the sequential portion of the application. Thus, microprocessor performance gains are slowing down due to power and parallelism limitation.

Customized hardware accelerators in the form of specialized datapath and memory management promise orders-of-magnitude performance and energy gains compared to general-purpose cores. Though such accelerators are not new to the architecture community, accelerator-centric architectures that combine general-purpose cores with a rich set of customized accelerators onto the same die receive increasing attention in the literature. These on-chip accelerators are usually application-specific implementations of a particular functionality, and can range from simple tasks (e.g., a multiply-accumulate operation) to complex applications (e.g., medical imaging [CGG14b], database management [KGP13], Memcached [LAC14, LMS13]). Due to superior energy-efficiency, accelerators are increasingly common in mobile SoCs. It is reported that the number of specialized IP blocks across five generations of Apple SoCs has been growing consistently over the past decade [SRW15].

Moreover, the acquisition of Altera by Intel in 2015 promises continued development of closely-integrated Xeon+FPGA processors, which aims to provide customized accelerators for datacenters.

While such architectures promise tremendous performance/watt targets, system architects face a multitude of new problems, including but not limited to 1) how customized accelerators physically interconnect with cores and caches, 2) how to logically integrate customized accelerators into the existing memory hierarchies and operating systems, and 3) how to efficiently offload algorithm kernels from general-purpose cores to customized accelerators. A *poor* design at any level that does not consider the characteristics of customized accelerators can significantly limit the overall efficiency of the system.

1.1 Focus of This Dissertation: Accelerator-Centric Architectures

This dissertation focuses on performance and energy efficiency of accelerator-centric architectures. We observe that the integration of customized accelerators into conventional microprocessors requires close scrutiny, since it determines our ability to reap the benefit of customized accelerators.

1.1.1 Key Challenges for Integrating Customized Accelerators

First, most modern microprocessors use packet-switched network to interconnect between cores, cache banks, memory controllers etc. It enables high bandwidth and fairness by sharing channels to multiple packet flows. We observe that customized accelerators exhibit pairwise bulk transfer and streaming, causing massive contention in a short period and unbalanced usage of channels, which may prolong the overall latency of a packet-switched network.

Second, while most modern operating system (OS) ubiquitously use virtual memory to manage main memory capacity, physically addressed accelerators still require manual memory management. A unified virtual address space between the host core and customized accelerators can largely improve the programmability, which necessitates hardware support

for address translation. However, supporting address translation for customized accelerators with low overhead is nontrivial.

Third, as we improve the efficiency of hardware accelerators and on-chip efficiency, providing data to accelerators from off-chip sources remains challenging. Due to the slow scaling of processor pin-offs, moving data across the chip boundary will become a major performance and energy bottleneck. On the other hand, the scaling of memory and storage system has offered high internal bandwidth which is often left underutilized. Near-memory and near-storage accelerators attack these issues by moving accelerator closer to the data, thus avoiding data movement. However, their applications are limited by the power and area constraints of the compute engine and the requirement that the working set must fit in the attached memory module.

1.1.2 Goal of This Dissertation

In this dissertation, our goal is to develop efficient solutions to overcome the above challenges in integrating accelerators to conventional compute architectures.

We first propose the Hybrid network with Predictive Reservation (HPR) which prioritizes accelerator traffic using circuit-switching and meanwhile minimizes conflicts with global management. Our design is based on the observation that the memory access pattern of fixed-function accelerators follows highly predictable timing, so that on-chip resources such as circuit-switching paths can be scheduled speculatively.

Second, we observe that accelerators have regular page access behavior, exhibit page split phenomenon due to data tiling and are immensely sensitive to address translation latency. To remedy this, we propose implementing two-level TLBs for accelerators to reduce TLB misses and using the host page walk mechanism to reduce page walk latency.

Third, we propose a general acceleration platform called the Compute Hierarchy, offering on-chip, near-memory and near-storage acceleration capabilities. The compute hierarchy eliminates data movement and exposes high internal bandwidth by distributing hardware accelerators to memory modules within the conventional memory hierarchy. A global accel-

erator manager is introduced to enable flexible configuration. We demonstrate the effectiveness of the compute hierarchy by simulating an end-to-end web-scale content-based image retrieval pipeline.

The rest of the dissertation is organized as follows. Chapter 3 describes our new hybrid-switching NoC design. Chapter 4 presents our solution in supporting address translation for accelerator-centric architectures. Chapter 5 illustrates the compute hierarchy.

CHAPTER 2

Background and Related Work

This chapter serves as an overview of the state-of-the art accelerator research and recent advances in system integration.

2.1 Accelerator-Centric Architectures

Accelerator-centric architectures feature a rich set of accelerators specialized for various application domains. While accelerators serve as the primary computation power in such architectures, convention out-of-order cores, network-on-chip and memory hierarchy are still necessary to feed and control accelerators, and handle software portions that are control-heavy or non-parallelizable. The rest of this section characterizes accelerator-centric architectures based on coupling and details the design of Accelerator-Rich Architectures which is the baseline of discussions in the following chapters.

2.1.1 Loosely-Coupled Accelerators

We classify on-chip accelerators into two classes: 1) tightly coupled accelerators where the accelerator is a functional unit that is attached to the pipeline of a particular core - Garp [HW97], UltraSPARC T2 [NHW07], Intel's Larrabee [SCS08] are examples of this, and 2) loosely coupled accelerators where the accelerator is a distinct module attached to the on-chip interconnection network and can be shared among multiple cores. This dissertation focuses on loosely-coupled accelerators as they are more flexible and pose more challenges in integration.

Figure 2.1 shows an overview of an accelerator-rich architecture [CGG12a, CGG12b,

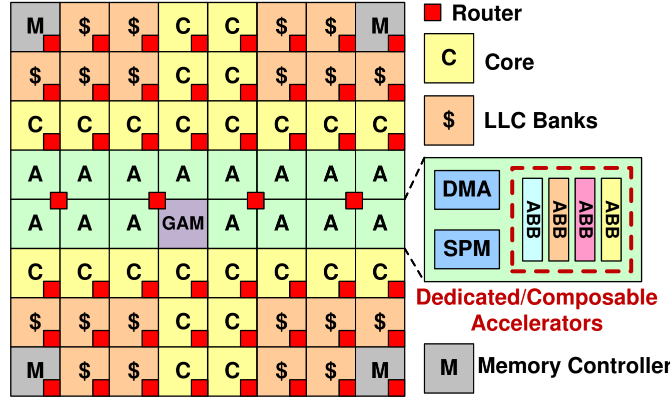

Figure 2.1: Overview of the Accelerator-Rich Architecture [CFG15]

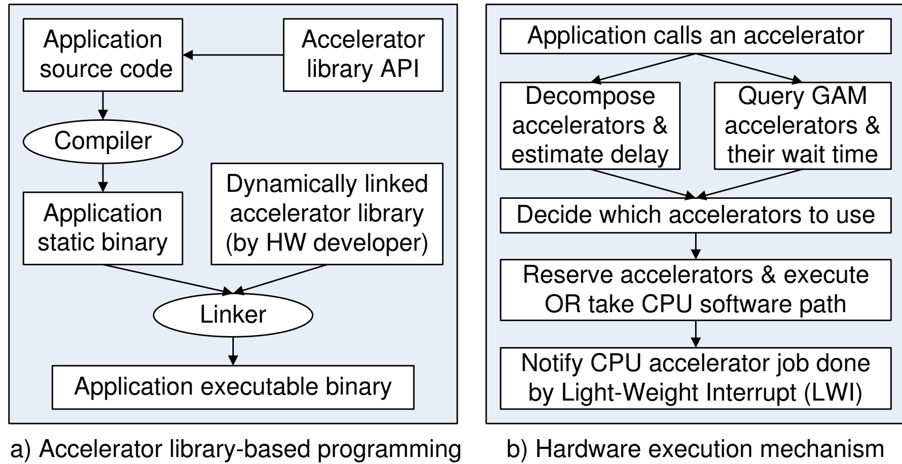

Figure 2.2: The programming model and underlying hardware execution mechanism [CFG15]

CGG13, CGG14a, CGH15], which is composed of cores, accelerators, the global accelerator manager (GAM), shared L2 cache banks, and shared network-on-chip (NoC). Each accelerator can be implemented at application-level or at kernel-level to collectively compose a more sophisticated functionality. To achieve high memory performance, each accelerator uses a user-managed scratch-pad memory (SPM) and communicates with the conventional cache hierarchy using a direct memory access (DMA) engine. The GAM is provided to interface between the cores and accelerators. The software programming model and hardware support for interacting with accelerators is presented in Figure 2.2. For each type of accelerator, a dynamic linked library (DLL) is provided, which will be linked to application code to handle calls to accelerators. From the hardware level, this DLL is supported collectively

using instruction set extension, the GAM and light-weight interrupt to avoid the inefficient interaction with the operating system.

There is a large amount of prior work that implements a coprocessor or accelerator specialized for a single application domain through either ASIC or FPGA technology. CHARM [CGG12b] explores the possibility of dynamically composing complex accelerators from accelerator building blocks. CAMEL [CGG13] extends CHARM by integrating FPGA logic on chip for better flexibility and longevity. Accelerator virtualization has been studied by works such as VEAL [CHM08], PPA [PPM09] and DySER [GHN12]. Efficient support for a unified address space has also been explored by [CFH17].

A number of recent work focuses on accelerating data center services on chip [LMS13, KGP13, CDL13]. TSSP [LMS13] proposes a cost-effective memcached deployment by implementing a memcached accelerator on an embedded SoC. Widx [KGP13] targets modern in-memory databases and implements a tightly-coupled accelerator exploiting the concurrency in indexing operations. While these proposals demonstrate performance advantages over conventional out-of-order cores on big data analytics, on-chip accelerators cannot avoid moving a large amount of data across memory hierarchies, which limits the gain from specialization.

Accelerator design for neural networks has become a major line of research in recent years. ASIC accelerators including DianNao [CDS14], DaDianNao [CLL14], Cnvlutin [AJH16], EIE [HLM16], Eyeriss [CES16, CKE17], Minerva [RWA16], Cambricon-X [ZDZ16], TETRIS [GPY17], SCNN [PRM17], and FPGA implementations such as NeuFlow [FMC11], SNNAP [MWN15], TABLA [MPA16], DnnWeaver [SPM16] and Caffeine [ZFZ16] have been proposed. While most of these works focus on the design of accelerator datapath and claim they support integration with frameworks such as Caffe [JSD14], the integration with conventional architecture is often not detailed.

2.1.2 Near-Memory Accelerators

Accelerators have also been attached to the memory bus or embedded into memory modules to process data from the main memory. Such accelerators are more coarse-grained and usually do not have frequent interaction with the host core since the communication cost is more significant. Virtual memory and address translation are typically not necessary as they can directly access their own physical memory.

Early work such as IRAM [PAC97] and CRAM [ESS99] integrate compute engines directly into DRAM modules. Although these architectures have very high bandwidth, they are limited to perform simple operations such as vector addition due to the design complexity and cost. More recently, Copacobana [KPP06] builds FPGA modules directly into DIMMs. NDA [FAM15] proposes stacking CGRA accelerators by inserting through-silicon vias to commodity DRAM DIMMs. AIM [CFG17] places FPGA modules between the DIMM and the memory network, making the design noninvasive to the existing memory controller, memory bus and DIMMs. Contutto [SRH17] prototypes such idea using a production POWER8 machine and demonstrates acceleration with end-to-end experiments.

Integrating accelerators with emerging memory technologies such as non-volatile memory has also been extensively studied. [YYD16] places hardware accelerators at the bottom layer of a Hybrid Memory Cube (HMC) to accelerate memory intensive operations including sorting, string matching, hash table lookup and memcopy. [AFH15] integrates a specialized reshape accelerator with 3D-stacked DRAM to accelerate data reorganization routines. PRIME [CLX16], ISAAC [SNM16] and PipeLayer [SQL17] propose to leverage the crossbar array structure of ReRAM to accelerate deep neural network applications.

2.1.3 Near-Storage Accelerators

The idea of offloading a portion of computation to near-storage compute engines is not new. IDISK [KPH98], proposed in 1998, adds embedded cores to each disk and utilizes serial communication links to form a interconnection network. With the advancement of flash technology, near-SSD computing attracts increasing attention in recent years. Projects such

as SmartSSD [DKP13], Active Disk [CPO13], Biscuit [GYB16] and Summariser [KMN17] propose to utilized the embedded cores in a modern SSD to avoid data movement and to free-up the host CPU and main memory. These works demonstrate the versatility of software, but also show the limitation of compute complexity and parallelism.

Incorporating reconfigurable hardware accelerators to distributed SSDs is also being investigated actively. Ibex [WIA14] offloads relational operators such as group-by to the FPGA coupled with each SATA SSD. IBM Netezza [SL11] offloads operations su ch filtering to the FPGA near storage. Willow [SGB14] adds programmability to the SSD interface to allow programmers to easily customize accelerators. QuickSAN [CS13], BlueDBM [JLL15], Caribou [ISA17] and Bluecache [XLJ16] deploy accelerators at the disaggregated storage servers to avoid data transmission over network. While most prior work focuses on accelerating big data analytics, ExtraV [LKY17] and GraFBoost [JWZ18] demonstrate significant speedup on out-of-memory graph processing.

2.1.4 Address Translation Support

Conventional closely-integrated accelerators adopt a separate physical address space from the host cores. In such a memory model, additional overhead in data replication and OS intervention in offloading is inevitable, which may diminish the gain of customization [CCF16]. Providing unified virtual address space between host cores and closely-integrated accelerators enables a variety of benefits, including the elimination of explicit data copying, increased performance of fine-grained memory accesses, support for cache coherence and memory protection. Efficient address translation support for accelerators is a crucial step towards a unified address space. We briefly overview the support for address translation on CPUs and GPUs, and discuss our proposed approach in Chapter 4.

Address Translation on CPUs. To meet the ever-increasing memory demands of applications, commercial CPUs have included one or more levels of TLBs [SGG12, Ham13] and private low-latency caches [BSS08, Int08] in the per-core MMU to accelerate address translation. These MMU caches, with different implementations, have been shown to greatly

increase performance for CPU applications [BCR10]. Prefetching [JM98, SDS00, KS02] techniques are proposed to speculate on PTEs that will be referenced in the future. While such techniques benefit applications with regular page access patterns, additional hardware such as a prefetching table is typically required. Shared last-level TLBs [BLM11] and shared MMU caches [Bha13] are proposed for multicores to accelerate multithreaded applications by sharing translations between cores. The energy overheads of TLB resources are also studied [KGC16], and advocate for energy-efficient TLBs. A software mechanism has also been proposed for address translation on CPUs [JM97].

Address Translation on GPUs. A recent IOMMU tutorial [KBB16] presents a detailed introduction to the IOMMU design within the AMD fused CPU-GPUs, with a key focus on its functionality and security. Though it also enables translation caching in devices, no detail or quantitative evaluation is revealed. To provide hardware support for virtual memory and page faults on GPUs, [PHB14, PHW14] propose GPU MMU designs consisting of post-coalescer TLBs and logic to walk the page table. As GPUs can potentially require hundreds of translations per cycle due to high parallelism in the architecture, [PHB14] uses 4-ported private TLBs and improved page walk scheduling, while [PHW14] uses a highly threaded page walker to serve bursts of TLB misses. ActivePointers [SBS16] introduces a software address translation layer on GPUs that supports page fault handling without CPU involvement. However, system abstractions for GPUs are required.

Current Virtual Memory Support for Accelerators. Some initial efforts have been made to support address translation for customized accelerators. The prototype of Intel-Altera heterogeneous architecture research platform [Int] uses a static 1024-entry TLB with 2MB page size to support virtual address for user-defined accelerators. A similar approach is also adopted in the design of a Memcached accelerator [LAC14]. Such a static TLB approach requires allocation of pinned memory and kernel driver intervention on TLB refills. As a result, programmers need to work with special APIs and manually manage various buffers, which can be a giant pain. Xilinx Zynq SoC provides a coherent interface between the ARM Cortex-A9 processor and FPGA programmable logic through the accelerator coherency port [NM13]. While prototypes in [CDL13, MWN15] are based on this platform, the address

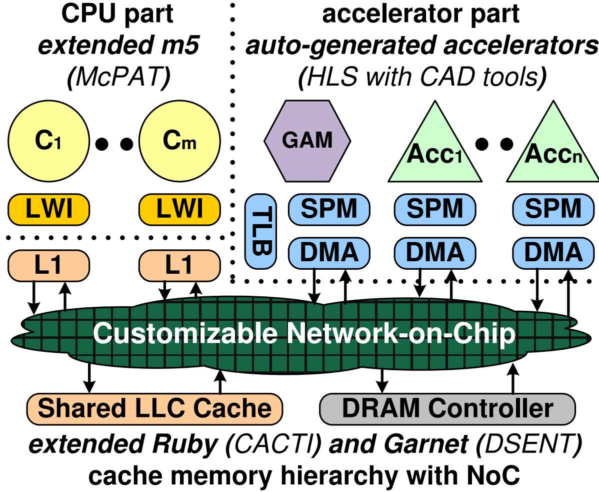

Figure 2.3: Overview of the PARADE simulator [CFG15]

translation mechanism is not detailed. [LMS13] assumes a system MMU support for the designed hardware accelerator.

Modern processors are equipped with IOMMUs [Adv11, Int14, KBB16] to provide address translation support for loosely coupled devices, including customized accelerators and GPUs. rIOMMU [MAB15] improves the throughput for devices that employ circular ring buffers, such as network and PCIe SSD controllers, but this is not intended for customized accelerators with more complex memory behaviors. With a unified address space, [OPH15] proposes a sandboxing mechanism to protect the system against improper memory accesses.

2.2 Simulation Infrastructure

Tools for designing and modeling accelerator-centric architectures have also been developed. PARADE [CFG15] provides a cycle-accurate full-system simulation platform for Accelerator-Rich Architectures. Figure 2.3 presents an overview of the PARADE simulator. On top of gem5 [BBB11], PARADE introduces the accelerator simulation modules that can be automatically generated through a high-level synthesis (HLS) description of the accelerator. With dedicated local SPM and DMA engine, PARADE adopts a load-compute-store three-stage accelerator execution model and task-level overlapping to achieve high performance.

Moreover, the GAM and light-weight interrupt is supported to enable efficient interactions between cores and accelerators. To design a new accelerator, users only need to provide a HLS C implementation. The automation tool chain will synthesize this C code to obtain clock frequency, pipeline initiation interval, pipeline depth, area and power for target ASIC design, which will be encoded into an accelerator simulation module. Meanwhile, a set of software APIs will be generated so that the new accelerator can be invoked from simulator benchmarks.

gem5-Aladdin [SSB16], also based on gem5, is able to model a cache-based accelerator with variable latency memory accesses and various optimization methods on DMA to improve efficiency. Unlike PARADE, gem5-Aladdin models accelerators based on a dataflow representation extracted from the profiling of the dynamic execution of a program, which enables fast design space exploration.

To provide higher efficiency than full-system simulation, ARACompiler [CCX15] proposes a highly automated design flow for prototyping accelerator-rich architectures and performing evaluation on FPGAs. Modern CPU-FPGA platforms such as Zynq [NM13] and Intel HARP [Int], which provide from embedded to server-like setups, are widely adopted in prototyping research ideas of on-chip accelerators.

CHAPTER 3

On-chip Interconnection Network for Accelerator-Rich Architectures

Accelerators offer orders-of-magnitude improvement in performance and energy efficiency as compared to general-purpose cores. Driven by the need for energy efficiency, the industry has proposed incorporating accelerators other than general-purpose cores into a die. These on-chip accelerators are application-specific implementations of a particular functionality, and can range from simple tasks (i.e. a multiply-accumulate operation) to complex tasks (i.e. FFT, encryption/decryption). An example of existing accelerator architecture designs is the IBM wire-speed processor architecture [FXB10], which features special-purpose dedicated accelerators optimized for security, XML parsing, compression, etc. We believe that future computing servers will improve their performance and power efficiency via extensive use of accelerators.

Unlike GPUs, accelerators cannot hide memory latency through extensive multithreading. The throughput of an accelerator is often bounded by the rate at which the accelerator is able to interact with the memory system. For this reason, an open question is: How should the accelerators, cores, buffers, L2 caches and memory controllers should be interconnected for maximum efficiency? Existing interconnect designs for accelerator architectures use partial crossbar switches [FXB10, CX13], bus [SCS08] and packet-switched networks [BKA10, LHW12] to transfer data between memory and accelerators. In Larrabee [SCS08], a wide bi-directional ring bus is used to limit the communication latency. Bakhoda et al. [BKA10] propose using a packet-switched mesh interconnect to connect accelerators and memory interfaces.

Unfortunately, as the number of on-chip accelerators increases, crossbar switches will consume intolerable area overhead due to the inability to scale. The ring bus solution is not sufficient to avoid latency that reaches problematic levels when the number of nodes increases beyond a certain point. Conventional packet-switched networks enable network sharing by implementing pipelined routers, but suffer from increased network latency and energy overhead. Kumar et al. [KPK07] proposed express virtual channels (EVC) in which intermediate nodes are bypassed in order to remove the delay in the buffer write, virtual channel arbitration and switch arbitration. However, EVC is limited to small-hop data communication and incurs significant overhead due to credit management.

To combat this problem, circuit-switched fabrics for CMPs are explored [GDR05, WL03, WSR05]. Compared to packet-switched networks, circuit-switched networks can significantly lower the communication latency, but suffer from the long setup and teardown latency. To amortize the setup overhead, [JPL08] explore hybrid-switched networks where circuit-switched and packet-switched flits share the same fabric. However, the reconfiguration of circuit-switched paths is determined by a setup network which adds an additional plane to the network. [YZS14] propose a time-division multiplexing hybrid-switching NoC for CPU-GPU heterogeneous platforms. However, due to the lack of global decision, the performance gains very little and even degrades in some cases..

Our investigation shows that accelerator memory accesses exhibit pairwise bulk transfer and streaming, creating highly utilized but frequently changing paths between the accelerator and memory interfaces. Moreover, hardware accelerators feature well-defined I/O patterns following fixed timing, allowing for better resource scheduling. Previous circuit-switched networks do not work well because the setup decisions are made locally and are unaware of the long-term communication patterns or the characteristics of accelerators; this results in unnecessary setups/teardowns and infrequent use of circuit paths. In the meantime, frequent setup and teardown requests hurt the overall performance.

In this work we propose the Hybrid network with Predictive Reservation (HPR) which globally manages NoC resources and reserves paths based on the timing of accelerator memory accesses. The global accelerator manager identifies the location and timing of accelerator

memory accesses, reserves paths and resolves conflicts in advance to improve circuit utilization. Meanwhile, the circuit-switched paths are reserved at the granularity of NoC cycles, maximizing the benefit of circuit-switched paths while minimizing the interference caused to packet-switched traffic.

The key contributions of this work are:

- We analyze the common characteristics of accelerator communication patterns, which suggests opportunities and challenges in shared resource management.

- We extend the global accelerator manager to identify exact locations and precise timing periods of accelerator memory accesses, and predictively reserve a series of circuit-switched paths before the accelerator starts to execute.

- To support the global circuit-switching decisions, we propose a new hybrid network design that provides predictable performance for circuit-switched traffic, while minimizing the interference caused to packet-switched traffic.

3.1 Overview of the Baseline Architecture

In this section we describe our baseline accelerator architecture and on-chip interconnect. Fig. 3.1 illustrates the overall chip layout of the accelerator-rich architecture [CGG12a]. This platform consists of cores (with private L1 caches), accelerators, shared L2 banks, memory controllers, the global accelerator manager, and the on-chip interconnect. We assume a 2D mesh topology with memory controllers placed in the corners, similar to the topology used in the Tilera TILE64 [WGH07] and Intel's 80-core design [VHR08], since it provides a simple and scalable network.

Our on-chip accelerator architecture features three key elements: 1) on-chip accelerators that implement complex specialized fixed functionalities, 2) buffer in NUCA [CGG12c], a modified cache that enables allocation of buffers in the shared last-level cache, 3) a global accelerator manager that assists with dynamic load balancing and task scheduling.

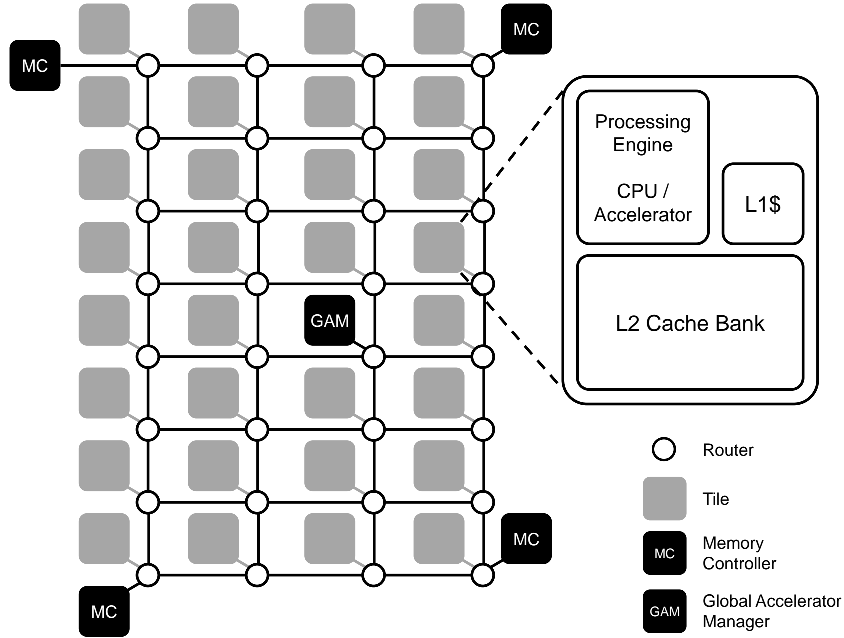

Figure 3.1: The overview of the accelerator-rich architecture

Loosely Coupled Accelerators: The on-chip accelerators that we consider in this work are specialized hardware computational units that are shared on an as-needed basis among multiple cores. These accelerators can significantly improve performance and save energy. However, this massive increase in performance typically comes with an increase in memory demand that is significantly larger than general-purpose cores. Fig. 3.2 shows the block diagram of loosely coupled accelerators featuring a dedicated DMA-controller and scratch-pad memory for local storage. The DMA engine is responsible for transferring data between the SPM and shared L2 caches, and between SPMs in the scenario of accelerator chaining.

Bu er in NUCA: Accelerators are designed to work with private buffers. These buffers are introduced to meet two goals: 1) bounding and reducing the fluctuation in latency between memory accesses, and 2) taking advantage of reuse of data within the buffer. As the number of accelerators in a system grows, the amount of dedicated buffer space devoted to these accelerators grows as well. In order to make more efficient use of chip resources, previous work provides a mechanism to allocate these buffers in cache space [FFI11, CGG12c, LHW12]. When not used as buffers, this memory would instead be used as regular cache. While

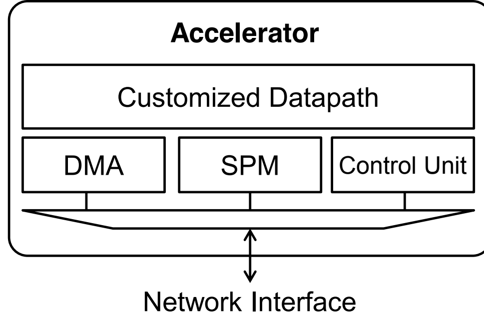

Figure 3.2: The microarchitecture of the on-chip accelerator

there are a number of mechanisms for allocating buffers in cache, we chose the Buffer in NUCA [CGG12c] scheme due to the consideration of spatial locality and the distributed nature of banked caches found in many-core systems.

The Global Accelerator Manager (GAM): The GAM fulfills two primary roles: 1) it presents a single interface for software to invoke accelerators, and 2) it offers shared resource management including buffer allocation and dynamic load balancing among multiple accelerators to allow collaboration on a single large task. To invoke an accelerator, a program would first write a description of the work to be performed to a region of shared memory. This description includes location of arguments, data layout, computation to be performed, and the order in which necessary operations to be performed. By evaluating the task description, the GAM splits the requested computation into a number of fixed-size data chunks to enable efficient parallelism, prescreens common TLB misses using a shared TLB, and then dispatches tasks to available accelerators [CGG14b].

3.2 Network Characterization

Due to the pipelined hardware implementation, accelerators often exhibit predictable memory access patterns such as bulk transfers and streaming. In this section we set out to characterize the common accelerator memory access patterns that motivate our proposed NoC design.

We collect the number of streaming flits between node pairs and show the network traffic

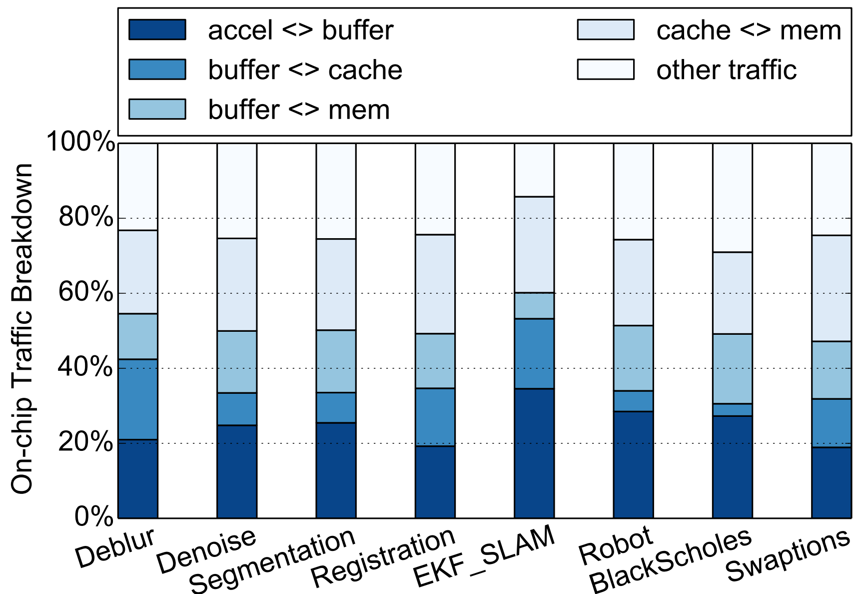

Figure 3.3: Network Traffic Breakdown. Results are collected on the baseline architecture described in Section 3.1. Details about the benchmarks and setups of the simulation platform can be found in Section 3.5.

breakdown in Fig. 3.3. Streaming flits are defined as consecutive flits traveling between the same source/destination pair. We can observe that the data streams account for a considerable fraction of total on-chip data traffic – more than 70% of total flits transmitted – which suggests that effective optimizations will have a significant impact on performance and efficiency.

We further investigate the memory access patterns of certain benchmarks to explore the potential optimization opportunities. Fig. 3.4(a) shows the L2 cache bank access traces for `Deblur` from the medical imaging suite [CSR11]. The `Deblur` accelerator takes in two 3D arrays, performs the Rician deconvolution and then outputs the 3D deconvoluted array. The results are collected to show the input stage of one accelerator execution using our simulation platform. Memory data is mapped to cache lines at the memory page granularity to allow for previous optimization techniques, such as hardware prefetching and page coloring, to be easily adapted. As we can see from this figure, the accelerator generates consecutive memory requests toward three L2 cache banks. Since these requests have ideal spatial and temporal

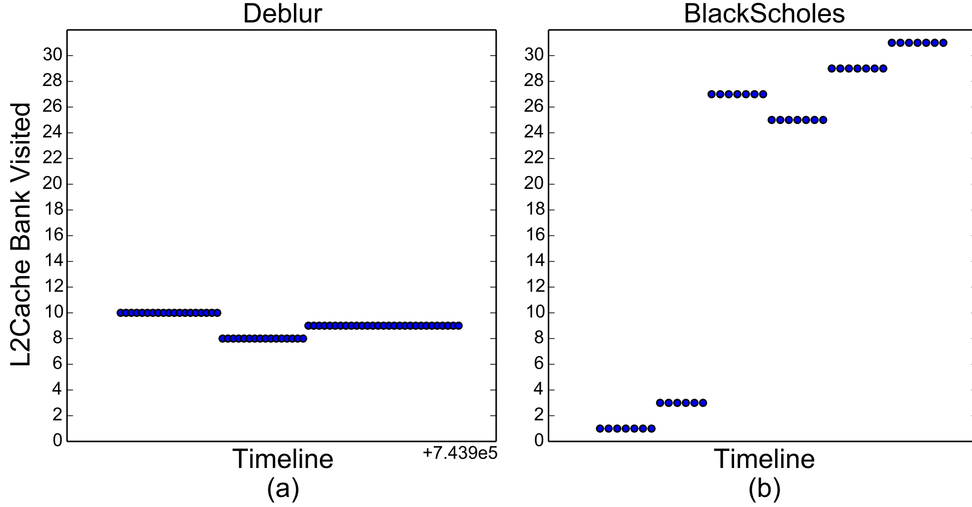

Figure 3.4: L2 cache bank access trace from the read phase of Deblur and BlackScholes. A read to a cache bank is depicted as a dot in the figure.

locality, setting up a circuit-switched path would greatly improve the throughput. Based on our observations, however, this is not always the case for accelerators targeting different domains.

Fig. 3.4(b) shows the access traces for BlackScholes from the financial analytics domain [Bie11]. The figure clearly shows that the accelerator reads from six different banks that correspond to six input arrays defined in the kernel. Unlike Deblur, the BlackScholes accelerator only requests a small amount of data from multiple cache banks. As a consequence, setting up circuit-switched paths for all destinations will not help to improve the overall performance since each path is not extensively used to amortize the setup overhead.

Although Deblur and BlackScholes have different memory access patterns, both cases demonstrate that requests are generated following predictable timings. Prior work also proved that accelerator data streams have constant timing intervals between consecutive requests [HZH11].

Based on the above observations, we summarize that data streams introduced by hardware accelerators often 1) account for a considerable portion of the total on-chip traffic, 2) exhibit uninterrupted streams with different lengths towards multiple destinations, and

3) generate memory requests following constant timing. Previous optimizations on packet-switched and hybrid-switched NoCs either add significant per-hop delay to support streaming or fail to identify the pattern to improve circuit utilization. In light of this, we propose global management combined with hybrid-switching to deliver efficient and predictable NoC performance to accelerator-rich architectures.

3.3 **HPR**: Hybrid Network and Predictive Reservation

We propose hybrid network with predictive reservation (HPR) to exploit the accelerator memory access characteristics to improve the network performance. The goal is to maximize the benefit of circuit-switching data streams while reducing the setup overhead and the interference with packet-switched traffic. To achieve this goal, we propose the global management to effectively identify the timing of setup and teardown of circuit-switching paths, and the time-division multiplexing (TDM) based hybrid-switching to provide efficient transmission with low overhead.

3.3.1 Global Management of Circuit-Switching

As described in Section 3.1, the global accelerator manager (GAM) provides shared resource management and dynamic load balancing. We extend the GAM to perform predictive reservation (PR) of NoC resources for accelerators to improve the throughput of the network.

The GAM is able to obtain information on the accelerator type and the read/write sets from the task description sent by a core before assigning the task to the accelerator. Therefore, the latency of the accelerator can be easily obtained and the possible read/write locations can be extracted by performing address translations. Based on this information, the communication pairs and the timing of communication are known to the GAM without actually executing the task.

Taking advantage of this knowledge, the GAM is capable of scheduling a series of circuit-switched paths at the granularity of time slot for the accelerator. As Fig. 3.5 shows, at

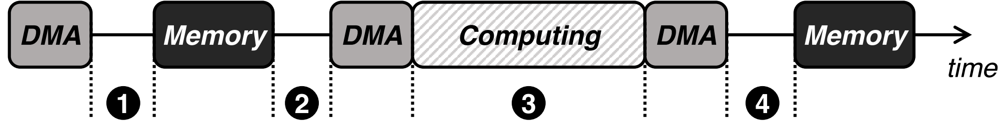

Figure 3.5: The service timeline of one accelerator execution

the beginning of processing the task, the DMA engine issues requests to memory interfaces, according to the read set, to fetch the data to the buffer. After these requests arrive at the memory interface, responses will be generated after the access latency, which can be estimated for cache and memory. Once all data requests are fulfilled, the accelerator will start to read from the input buffer and write to the output buffer following the fixed latency. Then results will be written back to the memory. As we can tell from the above process, the latency of each stage is either constant or can be estimated, which suggests that by identifying the timing of the four stages shown in Fig. 3.5, and in turn providing predictable circuit-switching transmission to traffic in each stage, unpredictable factors in the accelerator execution can be largely reduced.

To prevent conflicts in path reserving, the GAM tracks the current reservation status of each router in the network using a global reservation table. It searches the table for routes and conflicts before sending out the setup message. The XY-routing scheme is adopted for simplicity. If a conflict is found in the table, the GAM will delay the targeted time slots to the earliest available slots. Accordingly, future transactions will be delayed as an aftermath of the conflict. As we try to make reserved circuit-switched paths *as transient as possible*, the probability of conflicts is minimized, and the penalty of delayed reservation is also negligible. With the global management of circuit-switching, the result of setups can be guaranteed beforehand. Thus, no ACK message is required, thereby reducing network traffic and setup latency of circuit-switched paths.

In the event that a TLB miss occurs during the read/write session of an accelerator, the reserved circuit-switched paths must be voided since they can no longer match the communication period, leading to performance degradation. To combat this problem, we propose to translate the address from the read/write set at the GAM, buffer corresponding

TLB entries and then send those entries to the accelerator alongside the task description. As a result, the address translation requests at the accelerator side will always hit in the local TLB so that the accelerator is able to execute to completion without TLB misses. In other words, the TLB buffering mechanism eliminates the uncertainty in the accelerator execution, providing the GAM with better estimations of communication timings.

Example 1 shows a case for circuit-switched paths reservation to summarize the proposed global management scheme.

Example 1 Circuit-switched Paths Reservation

1: The GAM receives a task description from a core:
2: Perform bu er allocation
3: Extract memory addresses from read/write sets
4: **for all** physical address obtained from the read set **do**
5:     Locate the L2 $ bank and the Memory controller
6:     Try to reserve circuit-switched paths using XY-routing between the bu er, L2 cache bank and memory controller
7:     **if** con icts found **then**
8:         Delay the time window
9:     **end if**
10:     Update local reserved route record
11: **end for**
12: Reserve circuit-switching paths for both read and write session between the accelerator and the bu er
13: Do step 4 for the write set
14: Send the task description to the accelerator

3.3.2 Hybrid Network

We adopt a hybrid-switched router architecture similar to [JPL08, YZS14], as is shown in Fig. 3.6 and 3.7. To support the 2-stage hybrid-switched pipeline, the conventional 4-stage virtual-channeled wormhole router is extended with circuit-switched latches, a reservation

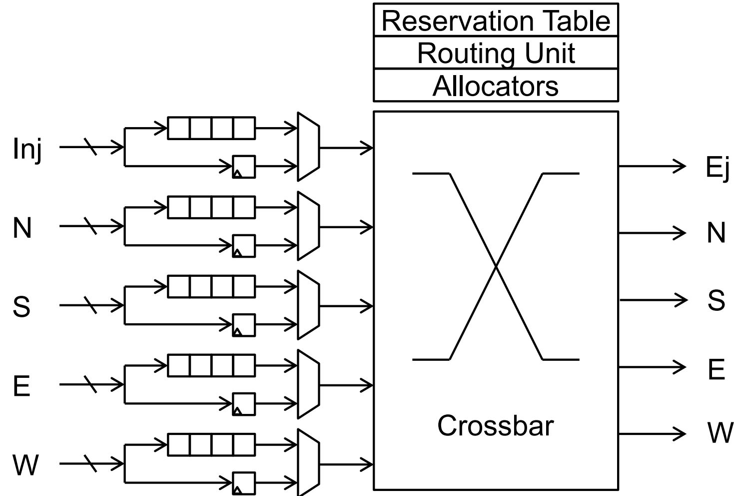

Figure 3.6: The hybrid-switched router architecture

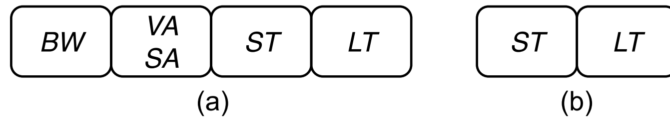

Figure 3.7: The packet-switched and circuit-switched pipeline

table, and demultiplexers.

Once the setup decision is made by the GAM, a setup flit carrying the reservation information will traverse the planned path. This setup flit reserves consecutive time slots for multiple future data streams sharing the same path (shown in Fig. 3.8). To avoid multiple table entries occupied by the same data stream, the reservation table is organized as each entry corresponds to a future data stream – where a start time, an end time, an input port and an output port are recorded. The router uses the reservation table to configure the switch in preparation for circuit-switching. By the completion of one circuit-switched session, the router recovers to packet-switching mode, and the old table entry is freed up to allow for new reservations.

By the start of a reserved circuit-switched session, the router needs to check the circuit field of the incoming flit to determine if a data stream is arriving. Once a data stream is confirmed, the router will forward incoming flits directly to the switch that was already

StartTime	EndTime	InPort	OutPort
200	208	1	3

→

StartTime	EndTime	InPort	OutPort
201	209	1	3

Figure 3.8: An example of the reservation tables from two neighboring routers

configured according to the reservation. The packet-switched flits in the buffer are not allowed to perform virtual channel allocation (VA) and switch allocation (SA) until the end of the session. If the circuit field is zero, this is a packet-switched flit which means no data stream matches the reservation. The reason for a missed reservation can be speculative schedules of circuit paths (i.e., miss-predictions of the data locations). As a result, the current circuit-switched session will be aborted and packet-switched flits are released to proceed to regular pipeline stages.

In order for data streams to *catch* the reserved session, the source and destination network interfaces are notified of the reserved window as well. The source node will not inject streaming flits to the NoC until the reserved session has taken place. This is essential for memory interfaces with variable latency (due to contention, scheduling policies, etc.) because by doing so, a *deadline* is set at the source node to bound the unpredictable memory latency. As a consequence, uninterrupted streams can be generated to fully use the reserved circuit-switched paths.

In summary, we propose HPR to effectively identify the timing of data streams and speculatively reserve circuit-switched paths to improve the throughput of the network. Meanwhile, we design a hybrid network that allow the reserved circuit-switched paths to tolerate variable memory latency and provide predictable performance with minimal interference to the packet-switched traffic.

3.4 Evaluation Methodology

We extended the full-system cycle-accurate Simics [MCE02] and GEMS [MSB05] simulation platform and modified GARNET [AKP09] to model the baseline architecture described

Table 3.1: System parameters for the baseline architecture

Component	Parameters
NoC	4x8 mesh topology, XY routing, wormhole switching, 4-stage router pipeline
Core	4 Ultra-SPARC III-i cores @ 2GHz
L1 Cache	32KB per core, 4-way set-associative 64B cache block, 3 cycle access latency pseudo-LRU replacement policy
L2 Cache	2MB shared, 32 banks, static-NUCA 8-way set-associative, 64B cache block 6-cycle access latency, pseudo-LRU
Coherence	MESI directory protocol
Memory	80GB/s bandwidth, 280-cycle access latency

in Section 3.1 and the proposed HPR scheme illustrated in Section 3.3. We use Orion 2.0 [KLP09] to estimate the power and energy consumption of the NoC. We consider a 32-node mesh topology, with one core, 30 on-chip accelerators and one GAM, with parameters shown in Table 3.1.

The benchmarks used in our study are four benchmarks from the medical imaging domain [CSR11], two from the financial analytics domain [Bie11], and three from the robotics domain [rob18]. We extract the computation-intensive kernel functions from these benchmarks and implement them as on-chip accelerators. We used the high-level synthesis tool Vivado HLS from Xilinx to generate the RTL of these computation-intensive kernel functions, and used the Synopsys design compiler to obtain ASIC characteristics of the accelerators.

We compare the HPR scheme against a baseline packet-switched NoC with four virtual channels, the express virtual channel (EVC) [KPK07] and a previously proposed TDM-based hybrid-switched NoC for heterogeneous platforms [YZS14] (denoted as HTDM below).

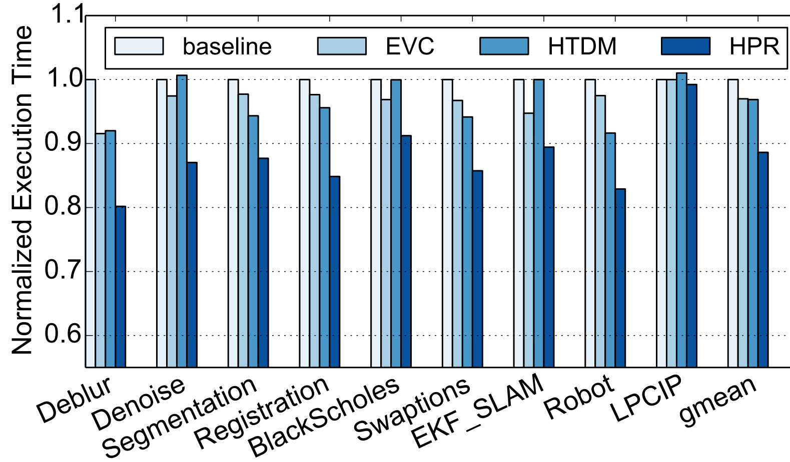

Figure 3.9: Normalized execution time

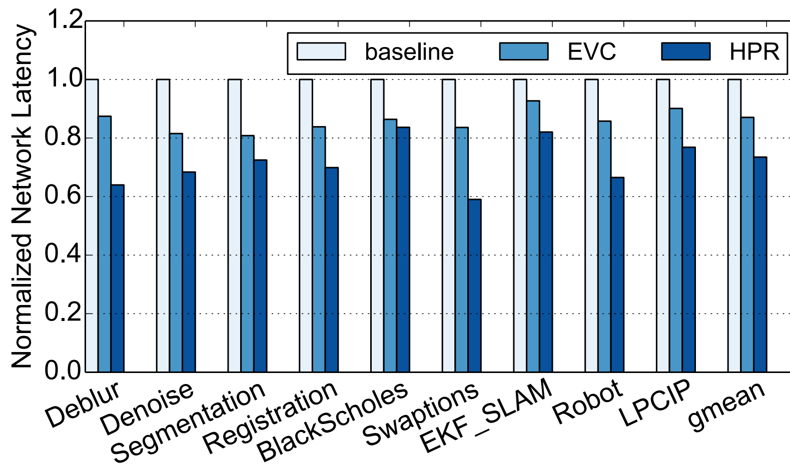

Figure 3.10: Normalized network latency

3.5 Experimental Results

Fig. 3.9 shows the comparison of normalized total execution time for each benchmarks against the baseline. HPR reduced the total execution time by 11.3% over the baseline, and 8.5% on average over EVC and HTDM. The largest reduction was seen for Deblur – about 20%. This is mainly attributable to high traffic levels between the accelerator and a small number of locations in the benchmark, leading to high circuit utilization. LPCIP, on the other hand, represents a relatively small read/write set with dynamic memory accesses defined during the execution. With GAM completely oblivious to them, the dynamic accesses cannot be covered by circuit-switched paths, and reserved paths also experience infrequent use due

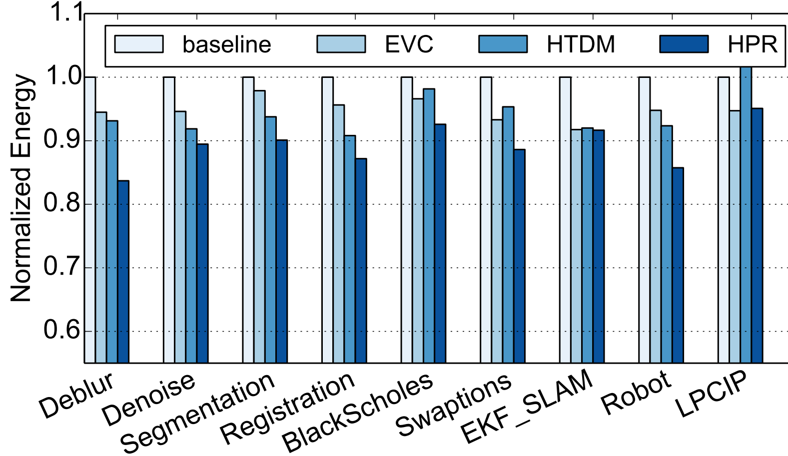

Figure 3.11: Normalized energy

to unpredictable delay. As can be seen from this figure, none of the evaluated schemes significantly improve the performance.

Fig. 3.10 shows the normalized network latency for EVC and HPR. EVC provides a consistent reduction in network latency of about 15% since the mechanism aims to improve the performance for all on-chip traffic and is oblivious to the heterogeneity in network traffic. Meanwhile, the goal of HPR is to provide efficient and predictable transmission for accelerator data streams. As a consequence, HPR reduces latency for the most critical part of the traffic, resulting in lower overall average network latency.

Fig. 3.11 shows the normalized network energy for the different benchmarks. Again, HPR shows energy reductions over the compared schemes – a reduction of around 11% on average going up to 16% for Deblur. In terms of LPCIP, whereas HTDM suffers from underutilized circuit-switched paths due to *short* dynamic accesses, HPR saves setup energy on dynamic accesses and still gains from predictively reserved circuit-switched paths.

In trying to study this gain further, we analyze the circuit-switching coverage of streaming flits with respect to the total number of on-chip streaming flits (shown in Fig. 3.12). As shown in this figure, both HTDM and HPR provide relatively high coverage of streaming flit. The reason that our scheme achieves better performance is essentially twofold. First, in setup of the same circuit-switched path, HTDM requires the confirmation of an ACK message before transmission, whereas our scheme eliminates the overhead by introducing global

Figure 3.12: Fraction of its covered by circuit-switched paths

Figure 3.13: Network latency of control messages

management. Second, our scheme also aims to identify and circuit-switch control messages (e.g., memory requests) which account for a small portion of its, but are critical to the overall latency of memory access. In contrast, previous schemes only target data movements. Fig. 3.13 illustrates this e ect by showing the normalized average network latency for control messages. The largest reductions were seen for Deblur, Swaptions and Robot, which receive the largest reductions in execution time as well. This is because these reductions in network latency will in turn lead to faster memory responses to the accelerator.

3.6 Summary

In this chapter, we propose the hybrid network with predictive reservation for accelerator-rich architectures to take advantage of common characteristics of accelerator communication patterns. The global management of NoC resources is proposed to predictively reserve circuit-switched paths for accelerators based on the knowledge of location and precise time period. The hybrid network is designed to provide e cient and predictable transmission during the reserved time period while reducing the interference caused to the packet-switched tra c. The evaluations show that HPR can achieve an average of 11.3% reduction in execution time compared to the baseline packet-switched network and 8.5% compared to EVC and a previous TDM-based hybrid-switched network.

CHAPTER 4

Supporting Address Translation for Accelerator-Centric Architectures

In light of the failure of Dennard scaling and recent slowdown of Moore's law, the computer architecture community has proposed many heterogeneous systems that combine conventional processors with a rich set of customized accelerators onto the same die [PPM09, HQW10, JAB10, SGG12, GHN12, KHL12, CGG12a, QHS13, ESC12, SYW13]. Such accelerator-centric architectures trade dark, unpowered silicon [EBS11, VSG10] area for customized accelerators that o er orders-of-magnitude performance and energy gains compared to general-purpose cores. These accelerators are usually application-speci c implementations of a particular functionality, and can range from simple tasks (e.g., a multiply-accumulate operation) to complex applications (e.g., medical imaging [CGG14b], database management [KGP13], Memcached [LAC14, LMS13]).

While such architectures promise tremendous performance/watt targets, system architects face a multitude of new problems, including but not limited to 1) how to integrate customized accelerators into the existing memory hierarchies and operating systems, and 2) how to e ciently o oad algorithm kernels from general-purpose cores to customized accelerators. One of the key challenges involved is the memory management between the host CPU cores and accelerators. For conventional physically addressed accelerators, if the application lives in the user space, an o oad process requires copying data across di erent privilege levels to/from the accelerator and manually maintaining data consistency. Additional overhead in data replication and OS intervention is inevitable, which may diminish the gain of customization [CCF16]. Zero-copy avoids copying bu ers via operating system support. However, programming with special APIs and carefully managing bu ers can be a

Figure 4.1: Problems in current address translation support for accelerator-centric architectures in an IOMMU-only con guration

giant pain for developers.

Although accelerators in current heterogeneous systems have limited support for virtual addresses, industry initiatives, such as the Heterogeneous System Architecture (HSA) foundation, are proposing a shift towards a uni ed virtual address between the host CPU and accelerators [HSA15]. In this model, instead of maintaining two copies of data in both host and device address spaces, only a single allocation is necessary. Consequently, an o oad process simply requires passing the virtual pointer to the shared data to/from the accelerator. This has a variety of bene ts, including the elimination of explicit data copying, increased performance of ne-grained memory accesses, support for cache coherence and memory protection.

Unfortunately, the bene ts of uni ed virtual address also come at a cost. A key requirement of virtually addressed accelerators is the hardware support for virtual-to-physical address translation. Commercial CPUs and SoCs have introduced I/O memory management units (IOMMUs) [Adv11, ARM15, Int14, KBB16] to allow loosely coupled devices to handle virtual addresses, as shown in Figure 4.1. These IOMMUs have I/O translation lookaside bu ers (IOTLBs) and logic to walk the page table, which can provide address translation support for customized accelerators. However, a naive IOMMU con guration cannot meet the requirement of today's high-performance customized accelerators as it lacks e cient TLB support, and excessively long latency is incurred to walk the page table on IOTLB misses.

Figure 4.2: Performance of the baseline IOMMU approach relative to ideal address translation

Figure 4.2 shows that the performance of the baseline IOMMU approach achieves only 12.3% of the ideal address translation, where all translation requests hit in an ideal TLB[1]; this leaves a huge performance improvement gap. Recent advances in IOMMU enable translation caching in devices [KBB16]. However, designing e cient TLBs for high-performance accelerators is nontrivial and should be carefully studied. Prototypes in prior studies encounter the challenge of virtual address support [LAC14, LMS13, CDL13, MWN15] as well. However, their focus is mainly on the design and performance tuning of accelerators, with either the underlying address translation approach not detailed or the performance impact not evaluated.

In this chapter, our goal is to provide an e cient address translation support for heterogeneous customized accelerator-centric architectures. The hope is that such a design can enable a uni ed virtual address space between host CPU cores and accelerators with modest hardware modi cation and low performance overhead compared to the ideal address translation.

By examining the memory access behavior of customized accelerators, we propose an

[1] Detailed experimental setup is described in Section 4.2. More analysis of this gap is presented in Section 4.3.1.

efficient hardware support for address translation tailored to the specific challenges and opportunities of accelerator-centric architectures that includes:

1. Private TLBs. Unlike conventional CPUs and GPUs, customized accelerators typically exhibit bulk transfers of consecutive data when loading data into the scratchpad memory and writing data back to the memory system. Therefore, a relatively small (16-32 entries) and low-latency private TLB can not only allow accelerators to save trips to the IOMMU, but can also capture the page access locality. On average, a private TLB with 32 entries can reduce 30.4% of the page walks compared to the IOMMU-only baseline, and improves the performance from 12.3% (IOMMU baseline) to 23.3% of the ideal address translation.

2. A Shared TLB. Data tiling techniques are widely used in customized accelerators to improve the data reuse within each tile and the parallelism between tiles. Due to the capacity limit of each accelerator's scratchpad memory, this usually breaks the contiguous memory region within a physical page into multiple data tiles that are mapped to different accelerator instances for parallelism. In light of this, we present a shared level-two TLB design to filter translation requests on common pages so that duplicate page walks will not be triggered from accelerator instances working on neighboring data tiles. Our evaluation shows that a two-level TLB design with a 512-entry shared TLB can reduce page walks by 75.8% on average, and improves the performance to 51.8% of the ideal address translation.

3. Host Page Walks. As accelerators are sensitive to long memory latency, the excessively long latency of page walks that cannot be filtered by TLBs degrades the system performance. While enhancing the IOMMU with MMU caches or introducing a dedicated MMU for accelerators are viable approaches [PHB14, PHW14], better opportunities lie in the coordination between the host core and the accelerators invoked by it. The idea is that by extending the per-core MMU to provide an interface, accelerators operating within the same application's virtual address space can offload TLB misses to the page walker of the host core MMU. This gives us three different benefits:

Figure 4.3: A detailed look at an accelerator connecting with system IOMMU (baseline)

rst, the page walk latency can be signi cantly reduced due to the presence of MMU caches [BCR10, BSS08, Int08] in the host core MMU; second, prefetching e ects can be achieved due to the support of data cache inasmuch as loading one cache line e ectively brings in multiple page table entries; third, cold misses in the MMU cache and data cache can be minimized since it is likely that the host core has already touched the data structure before o oading, so that corresponding resources have been warmed up. The experimental results show the host page walk reduces the average page walk latency to 58 cycles across di erent benchmarks, and the combined approach bridges the performance gap to 6.4% compared to the ideal address translation.

The remainder of this chapter is organized as follows. Section 4.1 characterizes address translation behaviors of customized accelerators to motivate our design; Section 4.2 explains our simulation methodology and workloads; Section 4.3 details the proposed design and evaluation; Section 4.4 discusses more use cases.

4.1 Characterization of Customized Accelerators

4.1.1 Accelerator-Centric Architectures

We present an overview of the baseline accelerator-centric architecture used throughout this work in Figure 4.1. In this architecture, CPU cores and loosely coupled accelerators share the

physical memory. Each CPU core has its own TLB and MMU, while all accelerators share an IOMMU that has an IOTLB inside it. A CPU core can launch one or more accelerators by o oading a task to them for superior performance and energy e ciency. Launching multiple accelerators can exploit data parallelism by assigning accelerators to di erent data partitions, which we call tiles.

The details of a customized accelerator are shown in Figure 4.3. In contrast to general-purpose CPU cores or GPUs, accelerators do not use instructions and feature customized registers and datapaths with deep pipelines [QHS13]. Scratchpad memory (SPM) is predominantly used by customized accelerators instead of hardware-managed caches, and data layout optimization techniques such as data tiling are often applied for increased performance. A memory interface such as a DMA (direct memory access) is often used to transfer data between the SPM and the memory system.

Due to these microarchitectural di erences, customized accelerators exhibit distinct memory access behaviors compared to CPUs and GPUs. To drive our design, we characterize such behaviors in the following subsections: the bulk transfer of consecutive data, the impact of data tiling, and the sensitivity to address translation latency.

4.1.2 Bulk Transfer of Consecutive Data

The performance and energy gains of customized accelerators are largely due to the removal of instructions through specialization and deep pipelining [QHS13]. To guarantee a high throughput for such customized pipelines|processing one input data every II (pipeline initialization interval) cycles, where II is usually one or two|the entire input data must be available in the SPM to provide register-like accessibility. Therefore, the execution process of customized accelerators typically has three phases: reading data from the memory system to the SPM in bulk for local handling, pipelined processing on local data, and then writing output data back to the memory system. Such bulky reads and writes appear as multiple streams of consecutive accesses in the memory system, which exhibit good memory page locality and high memory bandwidth utilization.

Figure 4.4: TLB miss behavior of BlackScholes

To demonstrate such characteristics, we plot the trace of virtual pages that trigger TLB misses in BlackScholes in Figure 4.4 (our simulation methodology and workloads are detailed in Section 4.2). We can see that the TLB miss behavior is extremely regular, which is di erent from the more random accesses in CPU or GPU applications. Since accelerators feature customized deep pipelines without multithreading or context switching, the page divergence is only determined by the number of input data arrays and the dimentionality of each array. Figure 4.5 con rms this by showing the TLB miss trace in a single execution of BlackScholes, which accesses six one-dimensional input arrays and one output array. In addition, we can see that TLB misses typically happen at the beginning of the bulky data read and write phases, followed by a large number of TLB hits. Therefore, high hit rates can be expected from TLBs with su cient capacity.

This type of regularity is also observed for a string-matching application and is reported to be common for a wide range of applications such as image processing and graphics [SXS15]. We think that this characteristic is determined by the fundamental microarchitecture rather than the application domain. Such regular access behavior presents opportunities for relatively simple designs that support address translation for accelerator-centric architectures.

Figure 4.5: TLB miss trace of a single execution from BlackScholes

4.1.3 Impact of Data Tiling

Data tiling techniques are widely used on customized accelerators, which group data points into tiles that are executed atomically. As a consequence, each data tile can be mapped to a di erent accelerator to maximize the parallelism. Also, data tiling can improve data locality for the accelerator pipeline, leading to an increased computation to communication ratio. This also enables the use of double (ping-pong) bu ering.

While the input data array could span several memory pages, the tile size of each input data is usually smaller than a memory page due to limited SPM resources, especially for high-dimensional arrays. As a result, neighboring tiles are likely to be in the same memory page. These tiles, once mapped to di erent accelerators, will trigger multiple address translation requests on the same virtual page. Figure 4.6 shows a simple example of tiling on a 32 32 input oat array with 16 16 16 tile size, producing 8 tiles in total. This example is derived from the medical imaging applications, while the sizes are picked for illustration purposes only. Since the rst two dimensions exactly t into a 4KB page, 32 pages in total are allocated for the input data. Processing one tile requires accessing 16 of the 32 pages. However, mapping each tile to a di erent accelerator will trigger 16 8 = 128 address translation requests, which is 4 times more than the minimum 32 requests. This duplication in address translation requests must be resolved so that additional translation service latency can be

Figure 4.6: Rectangular tiling on a 32 32   32 data array into 16   16   16 tiles. Each tile accesses 16 pages and can be mapped to a di erent accelerator for parallel processing.

avoided. The simple coalescing logic used in GPUs will not be su cient because concurrently running accelerators are not designed to execute in lockstep.

4.1.4 Address Translation Latency Sensitivity

While CPUs expose memory-level parallelism (MLP) using large instruction windows, and GPUs leverage their extensive multithreading to issue bursts of memory references, accelerators generally lack architectural support for ne-grained latency hiding. As discussed earlier, the performance of customized accelerators relies on predictable accesses to the local SPM. Therefore, the computation pipeline cannot start until the entire input data tile is ready. To alleviate this problem, double bu ering techniques are commonly used to overlap communication with computation|processing on one bu er while transferring data on the other. However, such coarse-grained techniques require a careful design to balance communication and computation, and can be ine ective in tolerating long-latency memory operations, especially page walks on TLB misses.

To further demonstrate latency sensitivity, we run simulations with varied address translation latencies added to each memory reference. Figure 4.7 presents the performance slowdown of LPCIP and the geometric mean slowdown over all benchmarks from additional

Table 4.1: Benchmark descriptions with input sizes and number of heterogeneous accelerators. [CGG14b]

Domain	Benchmark	Algorithmic Functionality	Input Size	Sub Accs[2]
	deblur	Total variation deconvolution		4
Medical	denoise	Total variation minimization	128 slices of images, each	2
Imaging	regist	Linear algebra and optimizations	image of size 128 128	2
	seg	Dense linear algebra, and spectral methods		1
Commercial	black	Stock option price prediction	256K sets of option data	1
from PARSEC	stream	Clustering and vector arithmetic	64K 32-dimensional streams	5
[BKS08]	swapt	Swaption prices by Monte Carlo simulation	8K sets of option data	4
Computer Vision	dispmap	Sums of absolute di erences computation and accumulation	64 64 image size, 8 8 window, 64 max disparity	4
	lpcip	Log-polar forward transformation of image patch around each feature	128K features from images of size 640 480	1
Computer Navigation	ekfslam	Partial derivative, covariance, and spherical coordinate computations	128K sets of sensor data	2
	robloc	Monte Carlo localization using probabilistic model and particle lter	128K sets of sensor data	1

Figure 4.7: Geometric mean slowdown over all benchmarks with varied address translation latencies, with LPCIP being the most sensitive benchmark

latency. In general, address translation latency within 8 cycles can be tolerated by double bu ering. Any additional latency beyond 16 cycles signi cantly degrades overall system performance. LPCIP shows the highest sensitivity to additional cycles among all benchmarks since the accelerator issues dynamic memory accesses during the pipelined processing which is beyond the coverage of double bu ering. While GPUs are reportedly able to tolerate 600 additional memory access cycles with a maximum slowdown of only 5% [HKW14], the performance of accelerators will be decreased by 5x with the same additional cycles.

Such immense sensitivity poses serious challenges to designing an e cient address translation support for accelerators: 1) TLBs must be carefully designed to provide low access latency, 2) since page walks incur long latency which could be a few hundred cycles, TLB structures must be e ective in reducing the number of page walks, 3) for page walks that cannot be avoided, page walker must be optimized for lower latency.

4.2 Evaluation Methodology

Simulation. We use PARADE [CFG15], an open-source cycle-accurate full-system simulator to evaluate the accelerator-centric architecture. PARADE extends the gem5 [BBB11]

simulator with high-level synthesis support [CLN11] to accurately model the accelerator module, including the customized data path, the associated SPM and the DMA interface. We use CACTI [MBJ09] to estimate the area of TLB structures based on 32nm process technology.

We model an 8-issue out-of-order X86-64 CPU core at 2GHz with 32KB L1 instruction and data cache, 2MB L2 cache and a per-core MMU. We implement a wide spectrum of accelerators, as shown in Table 4.1, where each accelerator can issue 64 outstanding memory requests and has double bu ering support to overlap communication with computation. The host core and accelerators share 2GB DDR3 DRAM on four memory channels. We extend PARADE to model an IOMMU with a 32-entry IOTLB [ABY10]. To study the overhead of address translation, we model an ideal address translation in the simulator with in nite-sized TLBs and zero page walk latency for accelerators. We assume that a 4KB page size is used in the system for the best compatibility. The impact of using large pages will be discussed in Section 4.4.2. Table 4.2 summarizes the major parameters used in our simulation.

Table 4.2: Parameters of the baseline architecture

Component	Parameters
CPU	1 X86-64 OoO core @ 2GHz 8-wide issue, 32KB L1, 2MB L2
Accelerator	4 instances of each accelerator 64 outstanding memory references Double bu ering enabled
IOMMU	4KB page, 32-entry IOTLB
DRAM	2GB, 4 channels, DDR3-1600

Workloads. To provide a quantitative evaluation of our address translation proposal, we use a wide range of applications that are open-sourced together with PARADE. These applications are categorized into four domains: medical imaging, computer vision, computer navigation and commercial benchmarks from PARSEC [BKS08]. A brief description of each application, its input size, and the number of di erent accelerator types involved is speci ed

in Table 4.1. Each application may call one or more types of accelerators to perform di erent functionalities corresponding to the algorithm in various phases. In total, we implement 25 types of accelerators[2]. To achieve maximum performance, multiple instances of the same type can be invoked by the host to process in parallel. By default, we use four instances of each type in our evaluation unless otherwise speci ed. There will be no more than 20 active accelerators; others will be powered o depending on which applications are running.

4.3 Design and Evaluation of Address Translation Support

The goal of this work is to design an e cient address translation support for accelerator-centric architectures. After carefully examining the distinct memory access behaviors of customized accelerators in Section 4.1, we propose the corresponding TLB and MMU designs with quantitative evaluations step by step.

4.3.1 Overview of the Baseline IOMMU Approach

Figure 4.2 shows the performance of the baseline IOMMU approach relative to the ideal address translation with in nite-sized TLBs and zero page walk latency. Since an IOMMU-only con guration requires each memory reference to be translated by the centralized hardware interface, performance su ers from frequent trips to the IOMMU. On one hand, benchmarks with large-page reuse distances, such as the medical imaging applications, experience IOTLB thrashing due to the limited capacity. In such cases, IOTLB cannot provide e ective translation caching, leading to a large number of long-latency page walks. On the other hand, while the IOTLB may satisfy the demand of some benchmarks with small page reuse distances, such as computer navigation applications, the IOMMU lacks e cient page walk handling; this signi cantly degrades system performance on IOTLB misses. As a result, the IOMMU approach achieves only an average of 12.3% relative to the performance of the ideal address translation, which leaves a huge performance gap.

[2]Two of the implemented accelerators are shared across di erent benchmarks as subaccs The Racian accelerator is shared by Deblur and Denoise; the Gaussian accelerator is shared by Deblur and Registration.

In order to bridge the performance gap, we propose to reduce the address translation overhead in three steps: 1) provide low-latency access to translation caching by allowing customized accelerators to store physical addresses locally in TLBs, 2) reduce the number of page walks by exploiting page sharing between accelerators resulted from data tiling, and 3) minimize the page walk latency by o oading the page walk request to the host core MMU. We detail our designs and evaluations in the following subsections.

4.3.2 Private TLBs

To enable more capable devices, such as accelerators, recent IOMMU proposals allow IO devices to cache address translation in devices [KBB16]. This reduces the address translation latency on TLB hits and relies on the page walker in IOMMU on TLB misses. However, the performance impact and design tradeo s are not scrutinized in the literature.

4.3.2.1 Implementation

TLB sizes. While a large TLB may have a higher hit rate, smaller TLB sizes are preferable in providing lower access latency, since customized accelerators are very sensitive to the address translation latency. Moreover, TLBs are reportedly power-hungry and even TLB hits consume a signi cant amount of dynamic energy [KGC16]. Thus, TLB sizes must be carefully chosen.

Commercial CPUs currently implement 64-entry per-core L1 TLBs, and recent GPU studies [PHB14, PHW14] introduce 64-128 entry post-coalescer TLBs. As illustrated in Section 4.1.2, customized accelerators have much more regular and less divergent access patterns compared to general-purpose CPUs and GPUs; this fact endorses a relatively small private TLB size for shorter access latency and lower energy consumption. Next we quantitatively evaluate the performance e ects of various private TLB sizes. We assume a least-recently-used (LRU) replacement policy to capture locality.

a) Private TLB size for all benchmarks except medical imaging applications. Figure 4.8 illustrates that all seven evaluated benchmarks greatly bene t from adding private TLBs. In

Figure 4.8: Performance for benchmarks other than medical imaging with various private TLB sizes, assuming xed access latency.

general, small TLB sizes such as 16-32 entries are su cient to achieve most of the improved performance. The cause of this gain is twofold: 1) accelerators bene t from reduced access time to locally cached translations; 2) even though the capacity is not enlarged compared to the 32-entry IOTLB, accelerators enjoy private TLB resources rather than sharing the IOTLB. LPCIP receives the largest performance improvement from having a private TLB. This matches the observation that it has the highest sensitivity to address translation latency due to dynamic memory references during pipelined processing, since providing a low-latency private TLB greatly reduces pipeline stalls.

b) Private TLB size for medical imaging applications: Figure 4.9 shows the evaluation of the four medical imaging benchmarks. These benchmarks have a larger memory footprint with more input array references, and the three-dimensional access pattern (accessing multiple pages per array reference as demonstrated in Figure 4.6) further stresses the capacity of private TLBs. While apparently the 256-entry achieves the best performance of the four, the increased TLB access time would decease performance for other benchmarks, especially latency-sensitive ones such as LPCIP. In addition, a large TLB will also consume more energy.

Non-blocking design. Most CPUs and GPUs use blocking TLBs since the latency can be hidden with wide MLP. In contrast, accelerators are sensitive to long TLB miss latencies. Blocking accesses on TLB misses will stall the data transfer, reducing the memory bandwidth

Figure 4.9: Performance for medical imaging benchmarks with various private TLB sizes, assuming xed access latency.

utilization which results in performance degradation. In light of this, our design provides non-blocking hit-under-miss support to overlap TLB miss with hits to other entries.

Correctness issues. In practice, correctness issues, including page faults and TLB shootdowns [BRG89], have negligible e ects on the experimental results. We discuss them here for implementation purposes. While page faults can be handled by the IOMMU, accelerator private TLBs must support TLB shootdowns from the system. In a multicore accelerator system, if the mapping of memory pages is changed, all sharers of the virtual memory are noti ed to invalidate TLBs using TLB shootdown inter-processor interrupts (IPIs). We assume shootdowns are supported between CPU TLBs and the IOTLB based on this approach. We also extend IOMMU to send invalidations to accelerator private TLBs to ush stale values.

4.3.2.2 Evaluation

We nd that low TLB access latency provided by local translation caching is key to the performance of customized accelerators. While the optimal TLB size appears to be application-speci c, we choose 32-entry for balance between latency and capacity for our benchmarks, and also lower area and power consumption. On average, the 32-entry private TLB achieves 23.3% of the ideal address translation performance, one step up from the 12.3% IOMMU baseline, with an area overhead of around 0.7% to each accelerator. Further improvements

Figure 4.10: The structure of two-level TLBs

are possible by customizing TLB sizes and supporting miss-under-miss targeting individual applications. We leave these for future work.

4.3.3 A Shared Level-Two TLB

Figure 4.10 depicts the basic structure of our two-level TLB design, and illustrates two orthogonal bene ts provided from adding a shared TLB. First, the requested entry is previously inserted to the shared TLB by a request from the same accelerator. This is the case when the private TLB size is not su cient to capture the reuse distance so that the requested entry is evicted from the private TLB earlier. Speci cally, medical imaging benchmarks would bene t from having a large shared TLB. Second, the requested entry is previously inserted to the shared TLB by a request from another accelerator. This case is common when data tile size is smaller than a memory page, and neighboring tiles within a memory page are mapped to di erent accelerators (illustrated in Section 4.1.3). Once an entry is brought into the shared TLB by one accelerator, it is immediately available to other accelerators, leading to shared TLB hits. Requests that also miss in the shared TLB need to access IOMMU for page table walking.

4.3.3.1 Implementation

TLB size. While our design trades capacity for lower access latency in private TLBs, we provide a relatively larger capacity in the shared TLB to avoid thrashing. Based on the

evaluation of performance impact of private TLB sizes, we assume a 512-entry shared TLB for the four-accelerator-instances case, where LRU replacement policy is used. Though it virtually provides a 128-entry level-two TLB for each sharer, a shared TLB is more flexible in allocating resources to a specific sharer, resulting in improved performance.

Non-blocking design. Similar to the private TLBs, our shared TLB design also provides non-blocking hit-under-miss support to overlap TLB miss with hit accesses to other entries.

Inclusion policy. In order to enable the aforementioned use cases, entries requested by an accelerator must be inserted in both the private and the shared TLB. We adopt the approach in [BLM11] to implement a *mostly-inclusive* policy, where each TLB is allowed to make independent replacement decisions. This relaxes the coordination between private and shared TLBs and simplifies the control logic.

Placement. We provide a centralized shared level-two TLB not tied to any of the accelerators. This requires each accelerator to send requests through the interconnect to access the shared TLB which adds additional latency. However, we find that the benefit completely outweighs the added access latency for the current configuration. Much larger TLB sizes or more sharers (we will discuss this in Section 4.4.1) could benefit from a banked design, but such use cases are not well established.

Correctness issues. In addition to the TLB shootdown support in private TLBs, the shared TLB also needs to be checked for invalidation. The reason is in a mostly-inclusive policy, entries that are previously brought in can be present in both levels.

4.3.3.2 Evaluation

In contrast to private TLBs where low access latency is the key, the shared TLB mainly aims to reduce the number of page walks in two ways: 1) providing a larger capacity to capture the page locality for applications which is difficult to achieve in private TLBs without sacrificing access latency, 2) reducing TLB misses on common virtual pages by enabling translation sharing between concurrent accelerators.

Figure 4.11: Page walk reduction compared to the IOMMU baseline[3]

Figure 4.11 sheds light upon the page walk reduction[3] in our two-level TLB design. Compared to private TLBs only, adding a shared TLB consistently reduces the overall number of page walk requests. For medical imaging benchmarks, especially Deblur, Denoise and Registration, which suffer from insufficient private TLB capacity, the shared TLB significantly cuts the number of page walks by providing more resources. For benchmarks that already find enough entries in private TLBs, such as Segmentation and BlackScholes, the shared TLB reduces the number of page walks by decreasing TLB misses on common virtual pages, which are due to data tiling effects. In general, the two-level TLB design achieves 76.8% page walk reduction compared to the IOMMU-only approach, leaving only a small gap to an ideal two-level TLB. The 512-entry shared TLB only incurs around 0.3% area overhead to the four sharing accelerators. StreamCluster and DisparityMap involve multiple iterations over the same input data using different types of accelerators. The result shows a gap between the 512-entry case and the ideal case because the size of input data exceeds the reach of the 512-entry TLB, but has no problem fitting in the infinite-sized TLB which eliminates cold misses at the beginning of each iteration.

---

[3] Note that the number of page walks does not equal the number of translation requests even in the IOMMU case, since the IOTLB can filter part of them.

Figure 4.12: Page walk reduction from adding a 512-entry shared TLB to in nite-sized private TLBs

To further isolate the e ect of page sharing caused by data tiling, we run simulations with in nite-sized private TLBs so that the capacity issue is eliminated. Figure 4.12 shows the page walk reduction by adding a 512-entry shared TLB to in nite-sized private TLBs. As in nite-sized private TLBs leave only cold misses, the shared TLB exploits page sharing among those misses and lters duplicate ones, resulting in a 41.7% reduction on average. Notice that not all benchmarks greatly bene t from having a shared TLB, which is due to the di erent tiling mechanism of each application. While developing a TLB-aware tiling mechanism could potentially reduce duplicate TLB misses, it is not easy to do so when the input data size and tile size are user-de ned and thus can be arbitrary. We leave this for future work.

The remainder of the page walks are due to cold TLB misses, where alternating TLB sizes or organization can not make a di erence. Therefore, we propose an e cient page walk handling mechanism to minimize the latency penalty introduced by those page walks.

4.3.4 Host Page Walks

As the IOMMU is not capable of delivering efficient page walks, the performance of accelerators still suffers from excessive long page walk latency even with a reduced number of page walks. While providing a dedicated full-blown MMU support for accelerators could potentially alleviate this problem, there may not be a need to establish a new piece of hardware—especially when off-the-shelf resources can be readily leveraged by accelerators.

This opportunity lies in the coordination between the accelerators and the host core that launches them. After the computation task has been offloaded from the host core to accelerators, a common practice is to put the host core into spinning so that the core can react immediately to any status change of the accelerators. As a result, during the execution of accelerators, the host core MMU and data cache is less stressed; this can be used to service translation requests from the accelerators invoked by this core. By offloading page walk operations to the host core MMU, the following benefits can be achieved:

First, the MMU cache support is provided from the host core MMU. Commercial CPUs have introduced MMU caches to store upper-level entries in page walks [BCR10, Int08]. The page walker accesses the MMU cache to determine if one or more levels of walks can be skipped before issuing memory references. As characterized in Section 4.1.2, accelerators have extremely regular page access behaviors with small page divergence. Therefore, the MMU cache can potentially work very well with accelerators by capturing the good locality in upper levels of the page table. We expect that the MMU cache is able to skip all three non-leaf page table accesses most of the time, leaving only one memory reference required for each page walk. We assume an AMD-style page walk cache [BCR10] in this work, which stores entries in a data cache fashion. However, other implementations, such as Intel's paging structure cache [Int08], could provide similar benefits.

Second, PTE (page table entry) locality within a cacheline provides an opportunity to amortize the cost of memory accesses over more table walks. Unlike the IOMMU, the CPU MMU has data cache support, which means a PTE is first brought from the DRAM to the data cache and then to the MMU. Future access to the same entry, if missed in

the MMU cache, could still hit in the data cache with much lower latency than a DRAM access. More importantly, as one cache line could contain eight PTEs, one DRAM access for a PTE potentially prefetches seven consecutive ones, so that future references to these PTEs could be cache hits. While this may not bene t CPU or GPU applications with large page divergence, we have shown that the regularity of accelerator TLB misses could permit improvement through prefetching.

Third, resources are likely warmed up by previous host core operations within the same application's virtual address space. Since a uni ed virtual address space permits a close coordination between the host core and the accelerators, both can work on the same data with either general-purpose manipulation or high-performance specialization. Therefore, the host core operations could very well warm up the resources for accelerators. Speci cally, a TLB miss triggered by the host core brings both upper-level entries to the MMU cache and PTEs to the data cache, leading to reduced page walk latency for accelerators in the near future. While the previous two bene ts can also be obtained through any dedicated MMU with an MMU cache and data cache, this bene t is unique to host page walks.

4.3.4.1 Implementation

Modi cations to accelerator TLBs. In addition to the accelerator ID bits in each TLB entry, the shared TLB also needs to store the host process (or context) ID within each entry. On a shared TLB miss, a translation service request with the virtual address and the process ID is sent through the interconnect to the host core operating within the same virtual address space.

Modi cations to host MMUs. The host core MMU must be able to distinguish accelerator page walk requests from the core requests, so that PTEs can be sent back to the accelerator shared TLB instead of being inserted into the host core TLBs after page walking. As CPU MMUs are typically designed to handle one single page walk at a time, a separate port and request queue for accelerator page walk requests is required to bu er multiple requests. An analysis on the number of outstanding shared TLB misses is presented in

Section 4.4.1. Demultiplexer logic is also required for the MMU to send responses with the requested PTE back to the accelerator shared TLB.

Correctness issues. In contrast to implementing a dedicated MMU for accelerators where coordination with the host core MMU is required on page fault handling, our approach requires no additional support for system-level correctness issue. If a page walk returns a NULL pointer on the virtual address requested by accelerators, the faulting address is written to the core's CR2 register and an interrupt is raised. The core can proceed with the normal page fault handling process without the knowledge of the requester of the faulting address. The MMU is signaled once the OS has written the page table with the correct translation. Then, the MMU nishes the page walk to send the requested PTE to the accelerator-shared TLB. The support for TLB shootdowns works the same as in the IOMMU case.

4.3.4.2 Evaluation

To evaluate the e ects of host page walks, we simulate an 8KB page walk cache with 3-cycle access latency, and a 2MB data cache with 20-cycle latency. If the PTE request misses in the data cache, it is forwarded to the o -chip DRAM which typically takes more than 200 cycles. We faithfully simulate the interconnect delays in a mesh topology.

We rst evaluate the capability of the host core MMU by showing the average latency of page walks that are triggered by accelerators. Figure 4.13 shows that the host core MMU consistently provides low-latency page walks across all benchmarks, with an average of only 58 cycles. Given the latency of four consecutive data cache accesses is 80 cycles plus interconnect delays, most page walks should be a combination of MMU cache hits and data cache hits, with DRAM access only in rare cases. This is partly due to the warm-up e ects where cold misses in both MMU cache and data cache are minimized. Based on this, it is di cult for a dedicated MMU to provide even lower page walk latency than the host core MMU.

We further analyze the average translation latency of each design to relate to our latency sensitivity study. As shown in Figure 4.14(a), the average translation latency across all

Figure 4.13: Average page walk latency when offloading page walks to the host core MMU

benchmarks for designs with private TLBs and two-level TLBs is 101.1 and 27.7 cycles, respectively. This level of translation latency, if uniformly distributed, should not result in more than a 50% performance slowdown according to Figure 4.7. However, as shown in Figure 4.14(b), the average translation latency of the requests that trigger page walks is well above 1000 cycles for the two designs that use an IOMMU. This is due to both long page walk latency and queueing latency when there are multiple outstanding page walk requests. With such long latencies added to the runtime, accelerators become completely ineffective in latency hiding, even on shorter latencies which could otherwise be tolerated by double buffering. In contrast, host page walks reduce page walk latencies and meanwhile minimize the variance of address translation latency. Therefore, the overall performance benefits from a much lower average address translation latency (3.2 cycles) and decreased level of variations.

4.3.5 Summary: Two-level TLBs and Host Page Walks

Overall design. In summary, to provide an efficient address translation support for accelerator-centric architectures, we first enhance the IOMMU approach by designing a low-latency private TLB for each accelerator. Second, we present a shared level-two TLB design

Figure 4.14: Average translation latency of (a) all requests; (b) requests that actually trigger page walks

Table 4.3: Con guration of our proposed address translation support

Component	Parameters
Private TLBs	32-entry, 1-cycle access latency
Shared TLB	512-entry, 3-cycle access latency
Host MMU	4KB page, 8KB page walk cache [BSS08]
Interconnect	Mesh, 4-stage routers

to enable page sharing between accelerators, reducing duplicate TLB misses. The two-level TLB design e ectively reduces the number of page walks by 76.8%. Finally, we propose to o oad page walk requests to the host core MMU so that we can e ciently handle page walks with an average latency of 58 cycles. Table 4.3 summarizes the parameters of key components in our design.

Overall system performance. Figure 4.15 compares the performance of di erent designs against the ideal address translation. Note that the rst three designs rely on the IOMMU for page walks which could take more than 900 cycles. Our proposed three designs, as shown in Figure 4.15 achieve 23.3%, 51.8% and 93.6% of the ideal address translation performance, respectively, while the IOMMU baseline only achieves 12.3% of the ideal performance. The performance gap between our combined approach (two-level TLBs with host page walks)

Figure 4.15: Total execution time normalized to ideal address translation

and the ideal address translation is reduced to 6.4% on average, which is in the range deemed acceptable in the CPU world (5-15% overhead of runtime [BCR10, BGC13, BLM11, MCV08]).

4.4 Discussion

4.4.1 Impact of More Accelerators

While we have shown that signi cant performance improvement can be achieved for four accelerator instances by sharing resources including the level-two TLBs and host MMU, it is possible that resource contention with too many sharers results in performance slowdown. Speci cally, since CPU MMUs typically handle one page walk at a time, the host core MMU can potentially become a bottleneck as the number of outstanding shared TLB misses increases. To evaluate the impact of launching more accelerators by the same host core, we run simulations with 16 accelerator instances in the system with the same con guration summarized in Table 4.3.

Figure 4.16: The average number of outstanding shared TLB misses of the 4-instance and 16-instance cases

We compare the average number of outstanding shared TLB misses[4] for the 4-instance and 16-instance cases in Figure 4.16. Our shared TLB provides a consistent ltering effect, requiring on average only 1.3 and 4.9 outstanding page walks at the same time in the 4-instance and 16-instance cases, respectively. While more outstanding requests lead to a longer waiting time, subsequent requests are likely to hit in the page walk cache and data cache due to the regular page access pattern, thus requiring less service time. Using a dedicated MMU with threaded page walker [PHW14] could reduce the waiting time. However, the performance improvement may not justify the additional hardware complexity, even for GPUs [PHB14].

Figure 4.17 presents the overall performance of our proposed address translation support relative to the ideal address translation when there are 16 accelerator instances. We can see that even with the same amount of shared resource, launching 16 accelerator instances does not have a signi cant impact over the e ciency of address translation, with the overhead being 7.7% on average. While even more active accelerators promise greater parallelism, we

[4] As TLB misses are generally sparse during the execution, we only sample the number when there is at least one TLB miss.

Figure 4.17: Performance of launching 16 accelerator instances relative to ideal address translation

already observe diminishing returns in the 16-instance case, as the interconnect and memory bandwidth is saturating.

Another way of having more active accelerators in the system is by launching multiple accelerators using multiple CPU cores. However, the page walker in each core MMU will not experience higher pressure in such scenarios since our mechanism requires that accelerators only o oad TLB misses to the host core that operates within the same application's virtual address space. A larger shared TLB may be required for more sharers where a banked placement could be more e cient. We leave this for future work.

4.4.2 Large Pages

Large pages [TH94] can potentially reduce TLB misses by enlarging TLB reach and speedup misses by requiring less access to memory while walking the page table. To reduce memory management overhead, the OS with Transparent Huge Page [Arc10] support can automatically construct large pages by allocating contiguous baseline pages aligned at the large page size. As a result, developers no longer need to identify the data that could bene t from using large pages and explicitly request the allocation of large pages.

As we have shown that accelerators typically feature bulk transfers of consecutive data and are sensitive to long memory latencies, large pages are expected to improve the overall performance of accelerators by reducing TLB misses and page walk latencies. We believe this approach is orthogonal to ours and can be readily applied to the proposed two-level TLBs and host page walk design. It is worthwhile to note that the page sharing e ect that results from tiling high-dimensional data will become more signi cant under large pages, leading to an increased number of TLB misses on common pages. Our shared TLB design is shown to be e ective in alleviating this issue.

4.5 Summary and Future Work

The goal of this work is to provide simple but e cient address translation support for accelerator-centric architectures. We propose a two-level TLB design and host page walks tailored to the speci c challenges and opportunities of customized accelerators. We nd that a relatively small and low-latency private TLB with 32 entries for each accelerator reduces page walks by 30.4% compared to the IOMMU baseline. Adding a shared 512-entry TLB eliminates 75.8% of total page walks by exploiting page sharing resulting from data tiling. Moreover, by simply o oading page walk requests to the host core MMU, the average page walk latency can be reduced to 58 cycles. Our evaluation shows that the combined approach achieves 93.6% of the performance of the ideal address translation.

This work assumes that the host core, after invoking accelerators, is spin waiting for the completion of accelerator executions, which e ectively makes core resources readily available to accelerators. Although this practice ensures the host core wakes up immediately and is recommended by many acceleration platforms, it may not hold true in all circumstances. For example, users may use the host core to execute other portions of the application while accelerators are executing the o oaded tasks. In this case, the host core and accelerators can contend at the page walker and MMU cache, leading to potential performance degradations. Also, the OS may decide to switch or migrate the host thread, making it di cult for accelerators to utilize the host core resources. For future work, we will create such scenarios

and quantify the e ects in order to solve the above issues.

Appendix

4.A A Performance Analysis of Modern CPU-FPGA Platforms with a Uni ed Address Space

[5] In recent years, various CPU-FPGA acceleration platforms with diversi ed architectural features have been developed and leading cloud service providers have started to incorporate FPGAs in their datacenters [PCC14, ORK15, ama18], indicating that FPGAs could play an important role in datacenter computing. Table 4.4 shows a classi cation of state-of-the-art CPU-FPGA platforms based on their physical integration and memory models.

Virtual memory, which FPGAs typically sidestep due to its ine ciency, makes its way back with closer integration to CPU, enabling a uni ed virtual memory space between the host and the FPGA. While conventional PCIe-based FPGAs rely on physical addressing, IBM experiments with virtual memory support on PCIe-based CAPI [SBJ15]. Intel rst provides virtual memory on a budget using pinned memory on HARPv1, and then extends to full- edged support with two-level TLBs and a page table walker on HARPv2. These e orts not only provide virtual memory capabilities to the FPGA but also signi cantly simplify the interaction between the host and the FPGA. However, whether the bene t of having uni ed address space outweighs the address translation overhead remains unstudied.

To evaluate how architectural characteristics a ect the performance of modern CPU-FPGA platforms, We conduct our quantitative comparison of four state-of-the-art platforms: 1) the Alpha Data board that represents the conventional PCIe-based platform with separate address spaces, 2) IBM CAPI that represents the PCIe-based system with a uni ed address space, 3) Intel HARPv1 that represents the QPI-based system with a uni ed address space, and 4) HARPv2 that represents a hybrid PCIe (non-coherent) and QPI (coherent) based system with a uni ed address space.

[5]This is a collaborative work with Peng Wei and Young-kyu Choi

Table 4.4: Classi cation of modern CPU-FPGA platforms

	Separate Address Space	Uni ed Address Space
PCIe Peripheral Interconnect	Alpha Data [Xil17], Microsoft Catapult [PCC14], Amazon F1 [ama18]	IBM CAPI [SBJ15], CCIx [cci18]
Processor Interconnect	N/A	Intel HARPv1 [Int] (QPI), Convey HC-1 [Bre10] (FSB)
Hybrid	N/A	Intel HARPv2 (QPI, PCIe)

4.A.1 Overview of Modern CPU-FPGA Platforms

Typical PCIe-based CPU-FPGA platforms feature Direct Memory Access (DMA) and private device DRAM (Fig. 4.18(a)). To interface with external network and memory, a memory controller IP and a PCIe endpoint with a DMA IP are required to be implemented on the FPGA, in addition to user-de ned accelerator function units (AFUs). Fortunately, vendors have provided hard IP solutions to enable e cient data copy and faster development cycles. For example, Xilinx releases device support for the Alpha Data card [Xil17] in the SDAccel development environment [sda18]. As a consequence, users can focus on designing application-related AFUs and easily swap them into the device support to build customized CPU-FPGA acceleration platforms.

IBM integrates Coherent Accelerator Processor Interface (CAPI) [SBJ15] into its Power8 and future systems, which provides virtual addressing, cache coherence and virtualization for PCIe-based accelerators (Fig. 4.18(b)). A coherent accelerator processor proxy (CAPP) unit is introduced to the processor to act as a proxy that maintains coherence for the o -chip accelerator across its virtual memory space. On the FPGA side, IBM also supplies a power service layer (PSL) unit alongside the user AFU. The PSL handles address translation and coherency functions while sending and receiving tra c as native PCIe-DMA packets. With the ability of accessing coherent shared memory of the host core, device DRAM and memory controller become optional for users.

Intel HARPv1 [Int] brings the FPGA one step closer to the processor via QPI where

(a) PCIe-based Alpha Data platform

(b) PCIe-based CAPI platform

(c) QPI-based HARPv1

(d) HARPv2 with 1 QPI and 2 PCIe channels

Figure 4.18: A tale of four CPU-FPGA platforms

an accelerator hardware module (AHM) occupies the other processor socket in a 2-socket motherboard. By using QPI interconnect, data coherency is maintained between the last-level cache (LLC) in the processor and the cache on the FPGA side. As shown in Fig. 4.18(c), an Intel QPI IP that contains a 64KB cache is required to handle coherent communication with the processor, and a system protocol layer (SPL) is introduced to provide address translation and request reordering to the user AFU. Speci cally, a page table of 1024 entries, each associated with a 2MB page (2GB in total), is implemented in SPL, which will be loaded by the device driver during runtime. As a result, the addressable virtual memory is limited to 2GB for AFU and the page table isstatic without page faults. Nevertheless, this low-latency coherent interconnect has distinct implications for programming models and overall processing models of CPU-FPGA platforms.

HARPv2 co-packages the CPU and FPGA to deliver even higher bandwidth and lower

latency than discrete forms. As shown in Fig. 4.18(d), the communication between CPU and FPGA is supported by two PCIe Gen3 x8 and one QPI (UPI in Skylake and later) physical links, which are presented as virtual channels on the user interface. The FPGA logic is divided into two parts: the Intel-provided FPGA interface unit (FIU) and the user AFU. The FIU implements interface protocols for the three physical links, maintains a coherent 64KB cache and supports partial recon guration of the user AFU. Moreover, a memory properties factory for higher level memory services and semantics is supplied to provide a push-button development experience for end-users.

Address translation, among many other things, is fully supported. A translation pipeline is implemented with two-level TLBs and a page table walker, with the ability to produce one translation each cycle. The rst level TLB is a simple direct-mapped cache, private to each communication channel. The second level TLB is set-associative and shared by all channels. This design, combined with large pages, makes TLB mises extremely rare. The page table walker receives requests from the TLB when a translation is not present in the TLB and accesses host memory to walk the page table. Although this page table is constructed by software rather than the real system page table, this approach does remove the 2GB virtual memory size limit.

4.A.2 Experimental Methodology

To evaluate the performance impact of di erent interconnects and memory models, we measure the CPU-FPGA communication bandwidth and latency using microbenchmarks based on the Xilinx SDAccel SDK 2017.2 [sda18] for Alpha Data, Alpha Data CAPI Design Kit [IBM15] for CAPI, and Intel AALSDK 5.0.3 [Int16] for HARPv1 and HARPv2. Each microbenchmark consists of two parts: a host program and a computation kernel. Following each platform's typical programming model, we use the C language to write the host programs for all platforms, and describe the kernel design using OpenCL for Alpha Data, and Verilog HDL for the other three platforms. The hardware con gurations of Alpha Data, CAPI, HARPv1 and HARPv2 in our study are summarized in Table 4.5.

Table 4.5: Platform con gurations of Alpha Data, CAPI, HARP and HARP2

Platform	Alpha Data	CAPI	HARPv1	HARPv2
Host CPU	Xeon E5-2620v3@2.40GHz	Power8 Turismo@4GHz	Xeon E5-2680v2@2.80GHz	Xeon E5-2600v4@2.4GHz
Host Memory	64GB DDR3-1600	16GB DDR3-1600	96GB DDR3-1600	64GB DDR4-2133
FPGA Fabric	Xilinx Virtex 7@200MHz	Xilinx Virtex 7@250MHz	Intel Stratix V@200MHz	Intel Arria 10@400MHz [z]
CPU $ FPGA	PCIe Gen3 x8, 8GB/s	PCIe Gen3 x8, 8GB/s	Intel QPI, 6.4GT/s	1 Intel QPI + 2 PCIe Gen3 x8
Device Memory	16GB DDR3-1600	16GB DDR3-1600 [y]	N/A	N/A

[y] The device memory in CAPI is not used in this work.

[z] The user clock can be easily con gured to 137/200/273 MHz using the supplied SDK, in addition to max 400MHz frequency.

Figure 4.19: Alpha Data PCIe-DMA bandwidth breakdown

4.A.3 E ective Bandwidth

Alpha Data Traditional PCIe-based platforms like Alpha Data contain two communication phases: 1) DMA transfer between host memory and device memory, and 2) device memory access. We measure the e ective bandwidth with various payload sizes for both phases. The PCIe bandwidth is shown in Fig. 4.19 and the e ective bandwidth for both phases are illustrated in Fig. 4.20.

As we can see from Fig. 4.19, the e ective PCIe bandwidth (1.6GB/s) reaches merely 20% of the advertised bandwidth (8GB/s for Gen3 x8). That is, the expectation of a high DMA bandwidth over PCIe is far away from being ful lled.

Figure 4.20: E ective bandwidth of Alpha Data, CAPI, HARPv1 and HARPv2

Apart from the protocol overhead which limits the achievable e ective bandwidth to 6.8GB/s [Law14], we found substantial system overhead involved in the measurement, including host bu er allocation and host memory copy [Coo12]. Since the address space of the host and FPGA are separate, the FPGA cannot directly retrieve user data stored in a pageable (unpinned) space in the host memory. A page-locked (pinned) memory bu er must be allocated to serve as a staging area for PCIe transfer. As a result, when a PCIe-DMA transaction starts, a pinned bu er is rst allocated in the host memory, followed by a memory copy of pageable data to this pinned bu er. The data is then transferred from the pinned bu er to device memory via PCIe. These three steps|bu er allocation, host memory copy and PCIe transfer|are sequentially processed in Alpha Data, which signi cantly deceases the overall bandwidth.

Next, we quantitatively evaluate the large PCIe-DMA bandwidth gap step by step, with

results shown in Fig. 4.19. (1) The protocol overhead lowers the theoretical PCIe bandwidth to 6.8GB/s from the advertised 8GB/s [Law14]; (2) The PCIe-DMA setup overhead prevents the 6.8GB/s PCIe bandwidth from being fully exploited. As a result, the highest achievable e ective PCIe-DMA bandwidth without bu er allocation and host memory copy decreases to 5.2GB/s; (3) The memory copy between the pageable and pinned bu ers further degrades the PCIe-DMA bandwidth to 2.7GB/s; (4) The bu er allocation overhead degrades the nal e ective PCIe-DMA bandwidth to only 1.6GB/s. This is the actual bandwidth that end users can obtain.

CAPI. CPU-FPGA platforms that supports uni ed address space, such as CAPI, HARPv1 and HARPv2, allow the FPGA to retrieve data directly from the host memory. Such platforms therefore contain only one communication phase: host memory access through the interconnect. For the PCIe-based CAPI platform, we simply measure the e ective read and write bandwidths of its PCIe channel for a variety of payload sizes, as shown in Fig. 4.20.

Compared to the Alpha Data board, CAPI provides end users with a much higher e ective PCIe bandwidth (3.3GB/s vs. 1.6GB/s). This is because CAPI supports a uni ed address space between the host and the FPGA, eliminating the staging overhead in PCIe transfers.

It is worthwhile to note that we observe a dramatic fallo of the PCIe bandwidth at the 4MB payload size. This could be the side e ect of CAPI's memory coherence mechanism. CAPI shares the last-level cache (LLC) of the host CPU with the FPGA, and the data access latency varies signi cantly between LLC hit and miss. Therefore, one possible explanation for the fallo is that CAPI shares 2MB of the 8MB LLC with the FPGA. The payloads of which the sizes are not larger than 2MB can t into LLC, resulting in a low LLC hit latency that can be well amortized by a few megabytes of data. Nevertheless, when the payload size grows to 4MB and cannot t into LLC, the average access latency of the payload data will suddenly increase, leading to the observed fallo . With the payload size further growing, this high latency is gradually amortized and the PCIe bandwidth gradually reaches the maximum value.

HARPv1. HARPv1's CPU-FPGA communication involves only one step: host memory

access through QPI; therefore, we only measure a set of e ective read and write bandwidths for di erent payload sizes, as shown in Fig. 4.20. We can see that both the read and write bandwidths of HARPv1 (7.0GB/s, 4.9GB/s) are much higher than the PCIe bandwidths of Alpha Data and CAPI. Therefore, the QPI-based CPU-FPGA integration does demonstrate a higher e ective bandwidth than the PCIe-based integration. However, HARPv1's remote memory access bandwidths are still lower than those of Alpha Data's local memory access. That is, similar with CAPI, HARPv1 can possibly be outperformed by Alpha Data if an accelerator keeps reusing the data in the device memory as a cache for the host memory.

HARPv2. Since HARPv2 allows the user AFU to work on a higher frequency, we present the e ective bandwidths at 200MHz, which is same frequency as Alpha Data, and at 400MHz, which is the highest supported frequency. Fig. 4.20 illustrates the measured results. We can see that HARPv2 outperforms all the other CPU-FPGA platforms in terms of e ective bandwidth (20GB/s at 400MHz, 12.8GB/s at 200MHz). This is due to the fact that HARPv2 connects the CPU and the FPGA through three physical channels: one QPI channel and two PCIe channels, resulting in a much higher aggregated bandwidth.

4.A.4 E ective Latency

As described in Section 4.A.1, both HARPv1 and HARPv2 include a 64KB cache for coherence purposes, and the host memory access thus falls into the classic cache access pattern. We measure the hit time and miss latency of this coherent cache, as shown in Table 4.6.

Table 4.6: CPU-FPGA access latency in HARPv1

Access Type	Latency (ns)
Read Hit	70
Write Hit	60
Read Miss	avg: 355
Write Miss	avg: 360

As we can see from Table 4.6, the coherent cache has a much longer access latency compared to user-de ned BRAMs { 70ns (14 FPGA cycles) for read hit and 60ns (12 FPGA

cycles) for write hit. To understand the hit latency, We decompose the hit time into three phases { address translation, cache access and transaction reordering { and measure the elapsed time of each phase, as shown in Table 4.7.

Table 4.7: Hit latency breakdown in HARPv1

Access Step	Read Latency (ns)	Write Latency (ns)
Address Translation	20	20
Cache Access	35	35
Transaction Reordering	15	5

The data demonstrate a perhaps exorbitant price (up to 100% extra latency) paid for address translation and transaction reordering. Worse still, the physical cache access latency is still high { 35ns (7 FPGA cycles). Given this small but long-latency cache, it is extremely hard, if not impossible, for an accelerator to harness the caching functionality, because the user-de ned BRAMs are simply much larger and yet faster.

It is worth noting that the HARPv2 platform supports higher clock frequencies than HARPv1, and thus can potentially lead to a lower cache access latency. However, this does not fundamentally change the fact that the latency of accessing the coherent cache is still much longer than that of accessing the user-de ned BRAMs. Therefore, HARPv2 does not fundamentally improve the usability of the coherent cache.

4.A.4.1 Discussion

Our experimental results demonstrate that supporting a uni ed address space between the host and the FPGA signi cantly improves the e ciency of data transfer. Built on top of the same interconnect technology, CAPI is able to provide much higher e ective bandwidth (3.3GB/s) than Alpha Data (1.6GB/s) due to the elimination of the staging process. This would lead to performance improvement for various communication-bound applications such as matrix-vector multiplication.

Moreover, integrating FPGAs into the same virtual address space changes the way the

FPGA interacts with the host. Instead of passively receiving data upfront from the host and working on the local copy, the FPGA is now allowed to initiate host memory accesses and co-process the shared data. This on-demand access enables ne-grained data sharing between the host and the FPGA, and applications with complex sharing patterns to be deployed on CPU-FPGA platforms. [CCC16] accelerates the supermaximal exact match (SMEM) seeding algorithm fro DNA sequence alignment using the HARPv1 platform. The authors implement a tight host-FPGA integration where many processing engines as well as software threads are served as worker threads to maximize the FPGA and the CPU utilization and maximize overall throughput.

However, the address translation overhead must be carefully accounted as it is on the critical path of accessing memory. We have shown that nearly 30% of FPGA cache hit latency is paid on translating the address. The TLB miss penalty is expected to be much longer for FPGA because multiple host memory accesses are required to walk the page table. We do not observe TLB misses in our experiments due to the fact that the current design preloads the TLB using software and makes TLB misses extremely rare for applications with regular memory access patterns, such as matrix-matrix multiplication. We would like to further examine the TLB miss penalty in future work and evaluate the impact on practical applications.

CHAPTER 5

Compute Hierarchy: A Case for Near Data Processing

[1] While hardware accelerators promise a great deal of computational power and energy e ciency, providing data to these compute engines poses a serious challenge. We have demonstrated in Chapter 4 that long memory latencies due to TLB misses could result in a tenfold reduction to the overall performance. This will continue to be problematic when considering the inevitable and pervasive out-pacing of memory devices by compute engines, known as theMemory Wall [WM95]. As a result, data movement is expected to dominate future systems' execution time as well as chip power dissipation [KDK11].

Due to the slow scaling of chip pin-out, it is not uncommon that modern systems feature memory modules with collective bandwidth that is many times of the processor's ability to make use of the bandwidth. In an example system shown in Figure 5.1, the internal storage bandwidth can be as high as 34GB/s with only 4 PCIe-attached SSDs, whereas the host side PCIe bandwidth is only 16GB/s. This bandwidth gap becomes more substantial as we scale up the SSD units in the system. To overcome this issue, prior works have proposed near-memory and near-storage processing which distributes compute engines into memory modules so that the internal bandwidth can be exploited and moving data across the chip boundaries can be avoided.

While the performance improvement and energy saving of these systems are very high, their application is limited to functions or operations with speci c computational pattern and data size not exceeding the capacity of the attached memory module. These constraints hinder their e ectiveness on practical applications or big data application pipelines, where these distributed accelerators are only suitable to perform part of the work and must coor-

[1] This is a collaborative work with Nazanin Faraphour

Figure 5.1: The throughput gap between the host PCIe bandwidth and the distributed SSD bandwidth. This gap grows as we scale-up the SSD units in the system.

dinate with other compute engines and memory modules in the system for data movement.

In this work, we present the compute hierarchy that combines on-chip, near-memory and near-storage accelerators, matching the conventional memory hierarchy. Each compute level with the memory level attached to, provides distinct compute and memory capabilities, offering a broad spectrum of acceleration options. To enable e ective acceleration on various application pipelines, we propose a holistic approach to coordinating between each compute level and managing resources. We experimentally deploy a billion-scale content-based image retrieval system on the compute hierarchy, and demonstrate that a proper application mapping eliminates data movement and achieves signi cant overall performance improvement.

5.1 Compute Hierarchy

In this section, we describe the architecture and design of the compute hierarchy system, and how the compute hierarchy matches di erent characteristics of the conventional memory hierarchy to form a general distributed compute platform with various compute and memory capabilities. While our compute engines are based on FPGA accelerators, the compute

Figure 5.2: The accelerator-interposed memory architecture with the AIMbus which enables inter-DIMM communication.

hierarchy is not dependent on a speci c type of accelerator logic; they could be general-purpose cores, GPUs or CGRAs. This work focuses on FPGAs due to their improved performance per watt ratio versus CPUs and GPUs since low power is critical for near-memory and near-storage acceleration where power/thermal constraints are more stringent.

5.1.1 Accelerator Design

On-chip Accelerator. We start with the on-chip hardware accelerator attached to coherent shared cache. As we discussed in Chapter 3 and Chapter 4, on-chip hardware accelerators achieve performance improvement by using private scratchpad memory to exploit data locality and customized deep pipeline to minimize control overhead. Data transfer between the scratchpad memory and the shared cache is handled by a DMA module shared by all processing elements (PEs).

To enable closer logical integration to the host CPU, virtual memory capabilities are supported by implementing TLBs and page table walker for the accelerator. Also, on-chip accelerators often have higher clock frequencies and larger area to work with in order to keep up with the host core and cache hierarchy, as is the case in HARPv2 [Int16]. However, the acceleration is limited by memory access latency and bandwidth once the working set exceeds the on-chip cache capacity. As a consequence, on-chip cache-attached accelerators are most suitable for compute-intensive workloads or applications that require frequency interactions with the host core.

Near-Memory Accelerator. While the performance of on-chip accelerators are bounded by the main memory bandwidth when data have to be fetched from o -chip and little locality can be exploited, near-memory accelerator overcomes the limit of narrow memory channels and achieves low-latency and high-bandwidth memory access by moving the compute engine closer to the main memory. Our design is based on accelerator-interposed memory (AIM) [CFG17]. As shown in Figure 5.2, an AIM module is introduced to interface with the memory network and the commodity DRAM DIMM, making it a noninvasive design to existing components in the system. Each AIM module contains a embedded FPGA accelerator that can be customized and controlled by the host, an AIMbus interface that allows inter-DIMM communication, logic to accelerator commands coming from the memory channel, and logic to forward memory responses to local accelerator, remote accelerator via AIMbus or the host CPU via the memory channel.

This near-memory accelerator is generally treated as a coprocessor and executes only when an application kernel is launched by the host CPU on the AIM module. Once a kernel is launched, the host memory controller hands over the control of a DIMM to the AIM module connected to it. The AIM module e ectively enforces a closed-row policy when accessing the DRAM, so that the host memory controller can assume all rows are in precharge state when the control is handed back. This minimizes the amount of synchronization and data sharing that takes place between the host CPU and AIM modules, which is vital to the e ciency of near-memory acceleration.

As DRAMs and the memory channel are typically designed to o er high capacity and low-latency, near-memory accelerators must avoid complex designs to work with the tight timing and power requirements of the DRAM standard. As a result, the FPGA accelerators are less powerful than the on-chip ones, but have lower memory access latency and can achieve higher aggregated bandwidth through parallel processing using multiple instances. The near-memory accelerators are suitable for parallelizable tasks with a large footprint and a high memory bandwidth requirement.

Near-Storage Accelerator. Figure 5.1 illustrates the throughput gap between the host PCIe bandwidth and the distributed SSD bandwidth. This gap becomes even larger as we

Figure 5.3: We distribute an FPGA accelerator to each SSD unit in order to match the distributed bandwidth and utilize the large aggregated bandwidth.

scale up the number of SSD units in the server, each with more ash channels. Instead of improving the I/O interconnect throughput to bridge the gap, we move compute engines closer to the storage by connecting an FPGA accelerator to each SSD unit via a local PCIe link. In addition to the user AFU, the FPGA accelerator implements a host interface to receive accelerator commands, an SSD interface to transfer data from/to the local SSD unit, and a storage network interface to request data from remote SSD units and supply data to accelerator peers. The decoupled SSD interface and storage network interface allow accelerators to access remote SSDs without interfering with the accelerators connected to them.

Similar to near-memory accelerators, near-storage accelerators are treated as coprocessors attached to the storage. They are designed to handle an entire computation kernel, eliminating the need for costly synchronizations with the host CPU. The FPGA chosen here must work with the cost and power budget of the server with respect to the number of SSD units in the system. Near-storage accelerators work the best for applications with reduction operations, where the data size is ideally reduced by orders of magnitude before they are transfered to upper levels in the memory hierarchy.

5.1.2 Global Accelerator Manager

To e ciently coordinate between hardware accelerators in the compute hierarchy and free CPU cores from managing resources, we propose using a on-chip hardware-based global accelerator manager (GAM) to 1) receive job requests for accelerators from cores; 2) distribute tasks to available accelerators; 3) track the tasks currently running or waiting to run on accelerators, their start time and estimated execution time; 4) interrupt the host core when the requested job is completed.

With multiple levels of accelerators, the GAM invokes accelerators at di erent levels according to the job description sent from the host core. If a job involves multiple compute levels, the GAM breaks the job into tasks to be assigned to di erent levels and makes sure results produced by one compute level can be fed to other levels. For near-memory accelerators, the GAM forces a write back in order to send the input data that were previously cached to the accelerators in memory. For near-storage accelerators, the GAM initiates PCIe transfer to send input data to the SSD-attached accelerators.

The GAM enables coarse-grained pipelining between di erent compute levels by assigning a task to an accelerator at a time and allowing accelerators at di erent levels to work on di erent tasks. The accelerator tasks are intentionally designed to be small enough to exploit task-level parallelism but large enough to amortize the data transfer overhead. Also, the GAM assigns tasks from the next job to accelerators without waiting for all the tasks in the previous job to complete. This reduces idle time and improves the pipeline e ciency.

5.2 Case Study: Content-based Image Retrieval

Content-based image retrieval (CBIR), which identi es relevant images from large-scale image databases based on the representation of visual content, has attracted increasing attention in recent two decades. We use CBIR as an application example where moving computation closer to data is vitally important. In this section, we present the challenges of implementing the on-line pipeline of CBIR and how mapping it to compute hierarchy

Figure 5.4: The general pipeline of billion-scale CBIR systems. In this work, we focus on components in the on-line stage { feature extraction, short-list retrieval, rerank and reverse lookup.

overcomes those challenges.

5.2.1 Application Overview

Figure. 5.4 shows the pipeline of CBIR. A typical billion-scale CBIR system consists of an o -line and an on-line stage. In the o -line stage, each database image is rst represented into a feature vector and then organized into clusters using indexing structures such as kd-trees or k-means. In the on-line stage, user query images are transformed into feature vector using the same method as in the o -line stage. The indexing structure generated in the o -line stage is leveraged to retrieve a short-list of candidate feature vectors that are likely to be close to the user query. These candidate vectors are then reranked based on their computed distance to the user query. Lastly, the original images corresponding to the k-nearest feature vectors are retrieved from the database.

In this work, we focus on the on-line stage of the CBIR pipeline as it is both data- and compute-intensive, and its throughput is crucial to user experience. We detail each component of the pipeline below.

Feature extraction. Image representation is key to the accuracy and performance of CBIR because it determines how e ciently the similarity between images can be measured. Since it is almost infeasible to directly compare images at pixel level, visual features within an image

are often extracted and packed into a fixed-length vector for image representation. While SIFT-based representations are commonly studied and adopted in early CBIR systems, the focus of recent studies have shifted to representations produced by deep neural network (DNN) with the explosive research on this area. Several works [RAS14, BSC14, BL15] have shown that DNN-based representations can significantly improve the recall accuracy on image retrieval benchmarks. In our implementation, we extract the feature vector from images using the VGGNet [SZ14] neural network and PCA compression with a dimensionality of 96. We assume user query inputs are sufficiently frequent for batched processing in order to improve the throughput of the system.

Short-list retrieval. To avoid exhaustive search on billion-scale databases, state-of-the-art CBIR systems extract a short-list of clusters that correspond to the centroids that are the closest to the query. These centroids are data points produced using clustering methods such as kd-trees or k-means during the off-line stage, so that the search space can be pruned at query time. Specifically, we compute the distances $\|q - C_k\|$ between the query and the centroids. Let us denote by $k_1; k_2; \ldots; k_K$ indices of K centroids that are produced by the indexing step during the off-line stage.

$$
\begin{aligned}
distances[k] &= \|q - C_k\|^2 \\
&= \|q\|^2 + \|C_k\|^2 - 2\langle q, C_k\rangle; \quad k = 1; \ldots; K
\end{aligned}
\tag{5.1}
$$

Note that the term $\|C_k\|^2$ in the decomposition can be precomputed, stored in RAM and reused for all queries. Therefore, the bottleneck of this step is the evaluation of two matrix-matrix multiplications, $\|q\|^2$ and $\langle q, C_k\rangle$, considering queries are batched. We then perform partial sorting of the distances array to produce the short-list for the query q.

Rerank. After short-list retrieval, we traverse the clusters and collects data points from these clusters to form a candidate list. Rerank computes the distances between the query and the data points within this candidate list. For both short-list retrieval and rerank steps, we use the square of Euclidean distance to measure the similarity between data points, which is defined as:

$$
\|p - q\|^2 = (p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2 \tag{5.2}
$$

Since short-lists are produced per query, the likelihood of di erent queries sharing the same clusters is small and thus batched processing is not applicable for most cases. As a result, the major computation in this step is vector-matrix multiplications. Lastly, a partial sorting on the computed distances is required to produce the K-nearest data points.

Reverse retrieval. To return the corresponding K-nearest images to the user, a reverse lookup in the image database is performed using the indices from the rerank results.

Billion-scale CBIR applications pose serious challenges on conventional CPU architectures and memory hierarchies. On one hand, exhaustive search on huge dataset that far exceeds the capacity of DRAM is impractical because (1) comparing against all data points stresses the overall throughput and latency; (2) the time and energy spent on bringing data to CPU outweigh those spent on computation. On the other hand, the performance of clustering methods degrades with increased dimensionality in the feature vector { a large portion of data blocks must be read in order to determine nearest neighbors, which is known as the curse of dimensionality [WSB98]. As a consequence, a large body of work focuses on compression methods such as binary codes and product quantization which reduces the dimensionality of feature vectors, leading to orders of magnitude reduction in data visited. However, these methods signi cantly penalizes the recall accuracy of the CBIR system.

5.2.2 Application Mapping

We target a CBIR system with 1 billion images. Each image is converted to a 96-dimension feature vector using VGG-16 [SZ14] neural network and PCA compression. All feature vectors are preprocessed using k-means algorithm, generating a list of cluster centroids. The incoming user query images are rst batched in the feature extraction and short-list retrieval steps, and then distributed according to their short-lists. Table 5.1 summarizes the memory requirement and the compute requirement of the major kernel within each CBIR pipeline stage.

The major algorithm involved in the feature extraction step is convolutional neural network, which is the most compute-intensive kernel while requiring the least amount of memory

Table 5.1: An overview of the memory requirement and the compute requirement of each CBIR pipeline stage.

	Memoy Requirement	Computation Requirement
Feature extraction	552MB, 11.3MB if compressed [HMD15] Neural network model parameters	High Convolutional neural network
Short-list retrieval	2.2GB Cluster centroids and cell info	Medium Matrix-matrix multiplications
Rerank	355GB 1 billion feature vectors	Low Matrix-vector multiplications
Reverse lookup	200TB - 2PB 1 billion images	Very low Database access

among all steps. The compressed neural network model parameters are only 11.3MB large using techniques proposed in [HMD15], which can t in on-chip SRAM. In light of this, the on-chip accelerator can potentially achieve higher performance because 1) it has higher frequency and more resources to exploit parallelism, 2) on-chip SRAM provides low access latency so that PEs can quickly be unblocked and resume processing. Our implementation is based on [ZLS15], where both computation and memory access are optimized for FPGA implementation to achieve high throughput across all layers.

For short-list retrieval, we use the decomposition in Equation 5.1 to turn this step to two matrix-matrix multiplications. The matrix-matrix multiplication kernel is ideally implemented using the on-chip accelerator due to its higher frequency and larger resources. However, since the inputkC_kk and precomputedkC_kk^2 takes 2.2GB of memory to store which exceeds the on-chip SRAM capacity, on-chip accelerators would need to frequently access o -chip DRAM and contend for the shared cache, leading to degraded performance. To this end, we support short-list retrieval by developing customized accelerators near memory. Although the compute power of near-memory accelerators is lower, the lower DRAM access latency and the ability to scale to multiple instances can o set this issue. Noted that near-storage accelerators can potentially provide even better scalability, but the mem-

Figure 5.5: Overview of the deployment of CBIR pipeline on the compute hierarchy

ory access latency is much higher which is undesirable for compute-intensive workloads like matrix-matrix multiplication.

The memory requirement of the rerank and the reverse lookup steps strongly favors near-storage computation { using either on-chip or near-memory acceleration would inevitably require moving large quantity of data across the memory hierarchy which is slow and energy-ine cient. Moreover, the performance of matrix-vector multiplication kernel is sensitive to memory bandwidth rather than computation power. Using near-storage acceleration would expose the large internal bandwidth of SSDs and can potentially achieve linear speedup as we scale-up the number of FPGA-SSD units in the system.

Figure 5.5 shows the overall deployment of the CBIR pipeline on the compute hierarchy system. The user query image batch is rst cached on-chip, and then converted to a batched of feature vectors by the on-chip feature extraction accelerators using the parameters stored entirely in on-chip SRAM. After the GAM is noti ed of the completion of the rst step, it hands the feature vector batch to the near-memory short-list retrieval accelerators. These accelerators performance matrix-matrix multiplications between the incoming feature vectors and cluster centroids stored entirely in DRAM. Then the GAM transfers individual query vector along with its retrieved short-list to the near-storage rerank accelerators. The rerank accelerators gathers dataset vectors from SSDs, computes distance to the query and perform

Table 5.2: Experimental setup of the compute hierarchy system

Component	Parameters
CPU	1 X86-64 OoO core @ 2GHz 8-wide issue, 32KB L1, 2MB shared L2
On-chip Accelerator	400MHz, 20GB/s bandwidth to shared cache
Near-Memory Accelerator	250MHz, 12.8GB/s bandwidth to DRAM and AIMbus
Near-Storage Accelerator	250MHz, 8.5GB/s bandwidth to SSD and network
PCIe-based Accelerator	250MHz, PCIe gen3 x16 link to the host CPU

a partial sort. Finally, the top K images are retrieved from the original image database and returned to the host. As we can see from the gure, the only data movement required is the user query vector. Other than that, the pipeline potentially touches several hundreds of data without actually moving any of them across the memory hierarchy.

5.3 Experimental Setup

To evaluate the performance of the compute hierarchy, we extend PARADE [CFG15] simulator to model accelerators with di erent frequencies and memory modules with various characteristics. We evaluate a system with on-chip accelerators attached to coherent share cache, accelerators attached to the main memory, and accelerators attached to SSDs. Our on-chip accelerator model is based on HARPv2, where a powerful 400MHz FPGA is coherently attached to CPU using one QPI link and two PCIe link [CCF16]. Our near-memory accelerator is modeled after AIM [CFG17] where a 250MHz FPGA is placed between each DRAM DIMM and the memory bus. An AIMbus connects each AIM module to enable inter-DIMM communication. The near-storage accelerator is con gured as a 250MHz FPGA with a PCIe link connecting to the SSD instance and a network link connecting to peer instances. We vary the number of PEs instantiated for on-chip accelerators. For near-memory and near-storage accelerators, we vary the number of FPGA instances in order to scale the number of DIMMs or SSD instances in the system. Table 5.2 summarizes the con guration of

the system.

We compare the compute hierarchy against common systems with discrete FPGA accelerators. This con guration is modeled closely after Amazon F1 [ama18] instances, based on our characterization of the bandwidth and latency on the real system [CCF16]. To make a fair comparison, we always assume the same number of FPGA accelerators in both cases, but all accelerators will share the same PCIe link in the discrete con guration.

For the CBIR pipeline simulation, we preprocess the image feature vectors with k-means to obtain 1000 cluster centroids. These centroids will later be used to retrieve the short-list for each query. In the nal rerank step, we compare each query against 1000 data points based on the short-list to make the simulation time manageable. Overall retrieval accuracy is evaluated using a software pipeline that implements exactly the same algorithms chosen by the hardware.

5.4 Experimental Results

5.4.1 Overall Performance

Figure 5.7 shows the runtime of the compute hierarchy as well as the PCIe-based acceleration. The baseline starts from one on-chip accelerator PE and one near-memory and near-storage accelerator instance, where the rerank step dominates the total execution time. As we increase the number of PEs or accelerator instances for each level, the compute hierarchy achieves increasing speedups on all three steps of the CBIR pipeline, with more than 7x overall speedup using 16 accelerator instances. The reason behind this scaling is that the compute requirement and memory requirement of the workload are perfectly matched by the characteristics of the memory hierarchy and the compute hierarchy attached to it. The performance keeps improving because the number of PEs is scaled up where the workload is bounded by computation and the memory interface is scaled up where the workload is bounded by communication.

While the PCIe-based acceleration also achieves speedup as we increase the number of

Figure 5.6: Runtime comparison between the PCIe-based acceleration and the compute hierarchy. Each total runtime is stacked by time spent on feature extraction, shot-list retrieval and rerank, from bottom to top, respectively.

PEs, the performance reaches a plateau after we have more than 4 PEs in the PCIe accelerator. This plateau clearly indicates the bandwidth bottleneck in the system, where the memory bandwidth cannot keep up with the compute engine and PEs are mostly starved. Unlike the compute hierarchy, the PCIe-based acceleration cannot scale memory bandwidth by adding more accelerator instances because the PCIe bandwidth would not improve. Overall, the compute hierarchy achieves 3x speedup using 16 accelerator instances compared to the PCIe-based acceleration with the same number of accelerators.

5.4.2 CBIR Pipeline Performance Breakdown

To demonstrate the e ectiveness of the compute hierarchy and the proposed application mapping, we implement each CBIR pipeline step using all types of accelerators and compare their performance.

Figure 5.7 illustrates the runtime of the feature extraction step implemented using PCIe-based, on-chip, near-memory or near-storage accelerators. Due to the compute-intensive nature of the workload, the on-chip accelerator has an clear advantage over others thanks to its higher operating frequency. Also, the performance of the on-chip accelerator keeps improving as the on-chip high-bandwidth and low-latency SRAM is able to feed more PEs.

Figure 5.7: Runtime comparison of implementing the feature extraction step using PCIe-based, on-chip, near-memory or near-storage accelerators.

Figure 5.8: Runtime comparison of implementing the short-list retrieval step using PCIe-based, on-chip, near-memory or near-storage accelerators.

On the contrary, PCIe-based, near-memory and near-storage accelerators show only limited performance scalability, as the long access latency to shared parameters start to o set the bene t of having more PEs.

A similar runtime comparison on the short-list retrieval step is illustrated in Figure 5.8. Although the major computation in this step is two matrix-matrix multiplications as we discussed in Section 5.2, the bottleneck shifts to the two vector-adds in Equation 5.1 as we dedicate more and more resource on matrix multiplication. As a result, the performance of a particular implementation is almost proportionate to its e ective memory bandwidth. The on-chip accelerator is bounded by the bandwidth of loading data from DRAM as they do not t in on-chip SRAM. Both the near-storage accelerator and PCIe-based accelerator are bounded by the PCIe bandwidth. The near-memory accelerator achieves better performance due to the support of AIMbus and higher aggregated DRAM bandwidth.

Figure 5.9: Runtime comparison of implementing the rerank step using PCIe-based, on-chip, near-memory or near-storage accelerators.

Figure 5.9 shows the rerank step runtime comparison of all implementations. As the input data can only be stored in the storage, PCIe-based, on-chip and near-memory accelerators all have to fetch data from the SSDs via PCIe. As a consequence, the performance is heavily limited by the I/O bandwidth, which is quickly saturated and adding more accelerators worsens the resource contention. Whereas the near-storage accelerators and the storage network allow us to expose the high internal bandwidth of SSDs, so that higher aggregated bandwidth can be easily achieved by scaling up the number of FPGA-SSD units in the system and each accelerator can be kept busy.

5.5 Discussion

We demonstrate the e ectiveness of the compute hierarchy using a practical CBIR system that can be properly mapped to all three levels of the compute hierarchy. In state-of-the-art software CBIR systems, throughput is sometimes favored over recall accuracy. Some prior work adopt techniques such as code book or product quantization to further compress the feature vector, so that the size of all database vectors can t in the main memory of a single server-class machine and the rerank step becomes more e cient with a smaller vector size [JTD11, BL16]. The downside, however, is the overall retrieval accuracy is signi cantly penalized by the feature vector compression.

To support a CBIR pipeline using approximate nearest neighbor methods, we map the

Figure 5.10: Execution time comparison of the three di erent algorithms deployed. From left to right is the compute hierarchy with exact vectors, compressed vectors and the linear search approach that skips the short-list retrieval step. The y-axis is in log scale.

feature extraction kernel to the on-chip accelerator, and map both short-list retrieval and rerank kernels to the near-memory accelerator, leaving the near-storage accelerators unused. Note that these kernels need to implement or be aware of the compression method, making them di erent than those in our evaluated pipeline. During runtime, the GAM makes sure the data ow from on-chip to near-memory accelerators but not to near-storage ones by invoking proper accelerators with the location of the input data in the memory.

At the other end of the tradeo is the linear search approach, where each query vector is compared against the entire list of data points using full vectors representation. This approach provides 100% accuracy and often serves as the ground truth in prior research. However, the amount of comparison and data movement required in this approach makes it prohibitive for software system implementations with quality-of-service constraints.

To deploy a CBIR pipeline with linear search, we simply skip the short-list retrieval step. After the feature extraction is completed by the on-chip accelerators, the GAM initiates PCIe transfer to directly push query vectors to the near-storage accelerators for linear scan.

Figure 5.10 shows the execution time comparison between the three algorithms deployed. The linear search approach takes almost 1000x longer to nish due to it exhaustive computation on the entire database. The execution time of the exact vector approach (compute

Figure 5.11: Execution time breakdown of the three di erent algorithms deployed.

hierarchy) is on par with the compressed vector approach, even though compressed vector allows tting all compressed vectors in DRAM and thus the computation is expected to be more e cient. The reason is that the compressed vector approach has to compare a lot more data points against each query in the rerank step in order to compensate for the overall retrieval accuracy, which o sets the saved execution time.

Figure 5.11 illustrates the execution time breakdown of the three algorithms deployed. The linear search approach is completely dominated by the rerank step as it does not have a short-list retrieval step and the data size of the rerank step is too big. The exact vector and compressed vector have more balanced distribution, while the compressed vector spends more time on rerank to improve accuracy. Overall, the retrieval accuracies of linear search, exact vector and compressed vector are 100%, 94% and 50% in the top 10 retrieved results.

In summary, the compute hierarchy o ers both distributed computational power which potentially eliminates data movement, and the exibility to adjust to the compute and memory requirements of di erent applications and di erent tradeo points within the same application pipeline.

CHAPTER 6

Conclusion

As traditional technology scaling slows, specialization becomes a viable solution for architects to keep improving performance and energy efficiency without relying on technology scaling. Customized accelerators, which inherently compromise generality in execution in order to gain efficiency, has been a growing topic in both academic research and industry development in the past decade. This work is motivated by our belief that future high performance computer systems will improve their performance and energy efficiency through extensive use of accelerators.

Integrating accelerators into conventional architectures is a challenging system and architecture problem, since hardware accelerators often exhibit very different behavior than general-purpose cores while architecture components such as caches and on-chip network are specifically designed to work efficiently with conventional cores, rather than accelerators. These issues, if not carefully examined, would hinder our ability to reap the performance and energy efficiency benefits promised by accelerators.

This dissertation is one of the first efforts in the computer architecture community to study the integration inefficiencies and propose to extend conventional system components to better support the characteristics of accelerators. Chapter 3 attacks from the physical integration angle, where we improve the traditional packet-switching on-chip interconnection network by introducing circuit-switching to prioritize accelerator traffic. Chapter 4 approaches the problem from the logical integration perspective, where we provide a unified address space between the host core and the accelerator by efficiently supporting address translation using two-level TLBs and host page walk. Chapter 5 focuses on the conventional memory hierarchy and integrates accelerators to match the characteristics of each level, cre-

ating a general-purpose acceleration system that o ers distinct computation and memory capabilities.

The approaches described in this dissertation demonstrate some initial steps towards e cient accelerator-centric architectures. A number of challenges remain to be addressed and new architectures and programming paradigms will be required to make this approach pervasive. Here we highlight some major challenges as our community moves forward:

While accelerators can potentially achieve signi cant performance improvement over general-purpose core, the interaction between the host core and the accelerator remains slow which application developers often sidestep. As a result, the accelerators are required to cover a su ciently large portion of the application in order to justify the acceleration e ort, which often limits the usability of accelerators. In addition to the issues discussed in detail in this dissertation, lots of research and engineering work on both hardware and software side, such as faster interconnection technologies, reduced software overhead by making the OS accelerator-aware, are still required to improve the interaction e ciency. This becomes even more challenging as we do not want e ciency at the cost of weakened safety guarantees.

Programmability has long been an issue of specialized accelerators. Based on a study conducted here at CDSC, today's most commonly adopted HLS ow still requires a considerable level of expertise in hardware to generate accelerators with speedup over multicores. We found that the ine ciency is largely due to the inability to describe memory operations which most high-level languages lack and the HLS tool is relied on to infer. As we have discussed in this dissertation, accelerators bene t from the customization on both computation and memory operations. As we aim to build more e cient accelerators in future accelerator-centric architectures, we need to expose the memory operations to developers through proper programming interfaces and make architectural improvements to make programming easier.

REFERENCES

[ABY10] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. \IOMMU: Strategies for Mitigating the IOTLB Bottleneck." In Workshop on Interaction between Operating Systems and Computer Architecture (WIOSCA), 2010.

[Adv11] Advanced Micro Devices, Inc. AMD I/O Virtualization Technology (IOMMU) Speci cation, 2011.

[AFH15] Berkin Akin, Franz Franchetti, and James C Hoe. \Data reorganization in memory using 3D-stacked DRAM." In ISCA-42, 2015.

[AJH16] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. \Cnvlutin: Ine ectual-neuron-free Deep Neural Network Computing." In ISCA-43, 2016.

[AKP09] N. Agarwal, T. Krishna, Li-Shiuan Peh, and N.K. Jha. \GARNET: A detailed on-chip network model inside a full-system simulator." In ISPASS, 2009.

[ama18] \Amazon EC2 F1 Instance." https://aws.amazon.com/ec2/instance-types/f1/ , 2018.

[Amd67] Gene M Amdahl. \Validity of the single processor approach to achieving large scale computing capabilities." In Proceedings of the April 18-20, 1967, spring joint computer conference, pp. 483{485. ACM, 1967.

[Arc10] Andrea Arcangeli. \Transparent hugepage support." In KVM Forum, 2010.

[ARM15] ARM Ltd. ARM System Memory Management Unit Architecture Speci cation, 2015.

[BBB11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. \The gem5 simulator." ACM SIGARCH Computer Architecture News, 39(2):1{7, 2011.

[BCR10] Thomas W. Barr, Alan L. Cox, and Scott Rixner. \Translation Caching: Skip, Don'T Walk (the Page Table)." In ISCA-37, 2010.

[BGC13] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. \E cient Virtual Memory for Big Memory Servers." In ISCA-40, 2013.

[Bha13] Abhishek Bhattacharjee. \Large-reach memory management unit caches." In MICRO-46, 2013.

[Bie11] Christian Bienia. Benchmarking Modern Multiprocessors. PhD thesis, Princeton University, January 2011.

[BKA10] Ali Bakhoda, John Kim, and Tor M Aamodt. \Throughput-e ective on-chip networks for manycore accelerators." In MICRO-43, 2010.

[BKS08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. \The PARSEC benchmark suite: characterization and architectural implications." In PACT-17, 2008.

[BL15] Artem Babenko and Victor Lempitsky. \Aggregating local deep features for image retrieval." In ICCV, 2015.

[BL16] Artem Babenko and Victor Lempitsky. \E cient indexing of billion-scale datasets of deep descriptors." In CVPR, 2016.

[BLM11] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. \Shared last-level TLBs for chip multiprocessors." In HPCA-17, 2011.

[Bre10] Tony M Brewer. \Instruction set innovations for the Convey HC-1 computer." IEEE Micro, (2):70{79, 2010.

[BRG89] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. \Translation Lookaside Bu er Consistency: A Software Approach." In ASPLOS-III, 1989.

[BSC14] Artem Babenko, Anton Slesarev, Alexandr Chigorin, and Victor Lempitsky. \Neural codes for image retrieval." In ECCV, 2014.

[BSS08] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. \Accelerating Two-dimensional Page Walks for Virtualized Systems." In ASPLOS-XIII, 2008.

[CCC16] Mau-Chung Frank Chang, Yu-Ting Chen, Jason Cong, Po-Tsang Huang, Chun-Liang Kuo, and Cody Hao Yu. \The SMEM Seeding Acceleration for DNA Sequence Alignment." In FCCM, 2016.

[CCF16] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. \A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms." In DAC-53, 2016.

[cci18] \Cache Coherent Interconnect for Accelerators." https://www.ccixconsortium.com/, 2018.

[CCX15] Yu-Ting Chen, Jason Cong, and Bingjun Xiao. \Aracompiler: a prototyping ow and evaluation framework for accelerator-rich architectures." In Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on, pp. 157{158. IEEE, 2015.

[CDL13] Eric S. Chung, John D. Davis, and Jaewon Lee. \LINQits: Big Data on Little Clients." In ISCA-40, 2013.

[CDS14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. \DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning." In *ASPLOS-XIX*, 2014.

[CES16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. \Eyeriss: A Spatial Architecture for Energy-e cient Data ow for Convolutional Neural Networks." In *ISCA-43*, 2016.

[CFG15] Jason Cong, Zhenman Fang, Michael Gill, and Glenn Reinman. \PARADE: A Cycle-Accurate Full-System Simulation Platform for Accelerator-Rich Architectural Design and Exploration." In *ICCAD*, 2015.

[CFG17] Jason Cong, Zhenman Fang, Michael Gill, Farnoosh Javadi, and Glenn Reinman. \AIM: accelerating computational genomics through scalable and noninvasive accelerator-interposed memory." In *MEMSYS*, 2017.

[CFH17] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinman. \Supporting address translation for accelerator-centric architectures." In *HPCA-23*, 2017.

[CGG12a] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. \Architecture Support for Accelerator-rich CMPs." In *DAC-49*, 2012.

[CGG12b] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. \CHARM: A Composable Heterogeneous Accelerator-rich Microprocessor." In *ISLPED*, 2012.

[CGG12c] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Chunyue Liu, and Glenn Reinman. \BiN: a bu er-in-NUCA scheme for accelerator-rich CMPs." In *ISLPED*, 2012.

[CGG13] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman. \Composable accelerator-rich microprocessor enhanced for adaptivity and longevity." In *ISLPED*, 2013.

[CGG14a] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. \Accelerator-Rich Architectures: Opportunities and Progresses." In *DAC-51*, 2014.

[CGG14b] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. \Architecture Support for Domain-Speci c Accelerator-Rich CMPs." *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):131:1{ 131:26, 2014.

[CGH15] Jason Cong, Michael Gill, Yuchen Hao, Glenn Reinman, and Bo Yuan. \On-chip interconnection network for accelerator-rich architectures." In *DAC-52*, 2015.

[CHM08] Nathan Clark, Amir Hormati, and Scott Mahlke. \VEAL: Virtualized execution accelerator for loops." In *ISCA-35*, 2008.

[CKE17] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. \Eyeriss: An energy-e cient recon gurable accelerator for deep convolutional neural networks." IEEE Journal of Solid-State Circuits, 52(1):127{138, 2017.

[CLL14] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. \Dadiannao: A machine-learning supercomputer." In MICRO-47, 2014.

[CLN11] J. Cong, Bin Liu, S. Neuendor er, J. Noguera, K. Vissers, and Zhiru Zhang. \High-Level Synthesis for FPGAs: From Prototyping to Deployment." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 30(4):473{491, April 2011.

[CLX16] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. \PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory." In ISCA-43, 2016.

[Coo12] Shane Cook. CUDA programming: a developer's guide to parallel computing with GPUs. Newnes, 2012.

[CPO13] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R Ganger. \Active disk meets ash: A case for intelligent ssds." In ICS, 2013.

[CS13] Adrian M. Caul eld and Steven Swanson. \QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks." In ISCA-40, 2013.

[CSR11] J. Cong, V. Sarkar, G. Reinman, and A. Bui. \Customizable Domain-Speci c Computing." IEEE Design Test of Computers, pp. 6{15, March 2011.

[CX13] Jason Cong and Bingjun Xiao. \Optimization of interconnects between accelerators and shared memories in dark silicon." In ICCAD, 2013.

[DKP13] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. \Query processing on smart SSDs: opportunities and challenges." In SIGMOD, 2013.

[EBS11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. \Dark Silicon and the End of Multicore Scaling." In ISCA-38, 2011.

[ESC12] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. \Neural acceleration for general-purpose approximate programs." In MICRO-45, 2012.

[ESS99] Duncan G Elliott, Michael Stumm, W Martin Snelgrove, Christian Cojocaru, and Robert McKenzie. \Computational RAM: Implementing processors in memory." IEEE Design & Test of Computers, 16(1):32{41, 1999.

[FAM15] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. \NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules." In HPCA-21, 2015.

[FFI11] Carlos Flores Fajardo, Zhen Fang, Ravi Iyer, German Fabila Garcia, Seung Eun Lee, and Li Zhao. \Bu er-integrated-Cache: a cost-e ective SRAM architecture for handheld and embedded platforms." In DAC-48, 2011.

[FMC11] Clement Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. \Neu ow: A runtime recon gurable data ow processor for vision." In CVPR Workshops, 2011.

[FXB10] Hubertus Franke, Jimi Xenidis, Claude Basso, Brian M Bass, Sandra S Woodward, Je rey D Brown, and Charles L Johnson. \Introduction to the wire-speed processor and architecture." IBM Journal of Research and Development, 54(1):3{1, 2010.

[GDR05] Kees Goossens, John Dielissen, and Andrei Radulescu. \ thereal network on chip: concepts, architectures, and implementations." IEEE Design Test of Computers, 22(5):414{421, 2005.

[GHN12] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. \DySER: Unifying Functionality and Parallelism Specialization for Energy-E cient Computing." IEEE Micro, 32(5):38{51, September 2012.

[GPY17] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. \Tetris: Scalable and e cient neural network acceleration with 3d memory." In ASPLOS-XXII, 2017.

[GYB16] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. \Biscuit: A framework for near-data processing of big data workloads." In ISCA-43, 2016.

[Ham13] Per Hammarlund. \4th generation Intel core processor, codenamed haswell." In Hot Chips, 2013.

[Hen17] John Hennessy. \The End of Road for General Purpose Processors and the Future of Computing." 2017.

[HKW14] Joel Hestness, Stephen W Keckler, and David A Wood. \A comparative analysis of microarchitecture e ects on cpu and gpu memory system behavior." In IISWC, 2014.

[HLM16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. \EIE: e cient inference engine on compressed deep neural network." In ISCA-43, 2016.

[HMD15] Song Han, Huizi Mao, and William J Dally. \Deep compression: Compressing deep neural networks with pruning, trained quantization and hu man coding." arXiv preprint arXiv:1510.00149, 2015.

[HQW10] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. \Understanding Sources of Ine ciency in General-purpose Chips." In ISCA-37, 2010.

[HSA15] HSA Foundation. HSA Platform System Architecture Speci cation 1.0, 2015.

[HW97] John R Hauser and John Wawrzynek. \Garp: A MIPS processor with a recon- gurable coprocessor." In FPT, 1997.

[HZH11] Rui Hou, Lixin Zhang, Michael C Huang, Kun Wang, Hubertus Franke, Yi Ge, and Xiaotao Chang. \E cient data streaming with on-chip accelerators: Opportunities and challenges." In HPCA-17, 2011.

[IBM15] IBM. Coherent Accelerator Processor Interface Users Manual Xilinx Edition, 2015. Rev. 1.1.

[Int] Intel Corporation. Accelerator abstraction layer software programmers guide.

[Int08] Intel Corporation. TLBs, Paging-Structure Caches, and Their Invalidation, 2008.

[Int14] Intel Corporation. Intel Virtualization Technology for Directed I/O Architecture, 2014.

[Int16] Intel. BDW+FPGA Beta Release 5.0.3 Core Cache Interface (CCI-P) Interface Speci cation, 9 2016. Rev. 1.0.

[ISA17] Zsolt Istvan, David Sidler, and Gustavo Alonso. \Caribou: intelligent distributed storage." Proceedings of the VLDB Endowment, 10(11):1202{1213, 2017.

[JAB10] C. Johnson, D.H. Allen, J. Brown, S. VanderWiel, R. Hoover, H. Achilles, C.-Y. Cher, G.A. May, H. Franke, J. Xenedis, and C. Basso. \A wire-speed powerTM processor: 2.3GHz 45nm SOI with 16 cores and 64 threads." In ISSCC, 2010.

[JLL15] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. \Bluedbm: An appliance for big data analytics." In ISCA-42, 2015.

[JM97] Bruce Jacob and Trevor Mudge. \Software-managed address translation." In HPCA-3, 1997.

[JM98] Bruce L. Jacob and Trevor N. Mudge. \A Look at Several Memory Management Units, TLB-re ll Mechanisms, and Page Table Organizations." In ASPLOS-VIII, 1998.

[JPL08] Natalie D Enright Jerger, Li-Shiuan Peh, and Mikko H Lipasti. \Circuit-switched coherence." In NOCS, 2008.

[JSD14] Yangqing Jia, Evan Shelhamer, Je Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. \Ca e: Convolutional architecture for fast feature embedding." In Proceedings of the 22nd ACM international conference on Multimedia, 2014.

[JTD11] Herve Jegou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. \Searching in one billion vectors: re-rank with source coding." In ICASSP, 2011.

[JWZ18] Sang-woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. \GraFBoost: Using accelerated ash storage for external graph analytics." In ISCA-45, 2018.

[KBB16] Andy Kegel, Paul Blinzer, Arka Basu, and Maggie Chan. \Virtualizing IO through IO Memory Management Unit (IOMMU)." In ASPLOS-XXI Tutorials, 2016.

[KDK11] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. \GPUs and the future of parallel computing." IEEE Micro, 31(5):7{17, 2011.

[KGC16] Vasileios Karakostas, Jayneel Gandhi, Adrian Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman S Unsal. \Energy-E cient Address Translation." In HPCA-22, 2016.

[KGP13] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsa , Kevin Lim, and Parthasarathy Ranganathan. \Meet the walkers: Accelerating index traversals for in-memory databases." In MICRO-46, 2013.

[KHL12] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. \Hardware Acceleration in the IBM PowerEN Processor: Architecture and Performance." In PACT-21, 2012.

[KLP09] Andrew B. Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. \ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-stage Design Space Exploration." In DATE, 2009.

[KMN17] Gunjae Koo, Kiran Kumar Matam, HV Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, Murali Annavaram, et al. \Summarizer: trading communication with computing near storage." In MICRO-50, 2017.

[KPH98] Kimberly Keeton, David A Patterson, and Joseph M Hellerstein. \A case for intelligent disks (IDISKs)." ACM SIGMOD Record, 27(3):42{52, 1998.

[KPK07] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. \Express Virtual Channels: Towards the Ideal Interconnection Fabric." In ISCA-34, 2007.

[KPP06] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfei er, and Manfred Schimmler. \Breaking ciphers with COPACOBANA{a cost-optimized parallel code breaker." In International Workshop on Cryptographic Hardware and Embedded Systems. Springer, 2006.

[KS02] Gokul B. Kandiraju and Anand Sivasubramaniam. \Going the Distance for TLB Prefetching: An Application-driven Study." In ISCA-29, 2002.

[LAC14] Maysam Lavasani, Hari Angepat, and Derek Chiou. \An FPGA-based in-line accelerator for Memcached." IEEE Computer Architecture Letters, 13(2):57{60, 2014.

[Law14] Jason Lawley. Understanding Performance of PCI Express Systems. Xilinx, 10 2014. Rev. 1.2.

[LHW12] Michael J Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. \The accelerator store: a shared memory framework for accelerator-based systems." ACM Transactions on Architecture and Code Optimization (TACO), 8(4):48, 2012.

[LKY17] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H Peter Hofstee, Gi-Joon Nam, Mark R Nutter, and Damir Jamsek. \ExtraV: boosting graph processing near storage with a coherent accelerator." Proceedings of the VLDB Endowment, 10(12):1706{1717, 2017.

[LMS13] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. \Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached." In ISCA-40, 2013.

[MAB15] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. \rIOMMU: E cient IOMMU for I/O devices that employ ring bu ers." In ASPLOS-XX, 2015.

[MBJ09] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. CACTI 6.0: A tool to model large caches. HP Laboratories, 2009.

[MCE02] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. \Simics: A full system simulation platform." Computer, 35(2):50{58, Feb 2002.

[MCV08] Collin McCurdy, Alan L. Coxa, and Je rey Vetter. \Investigating the TLB Behavior of High-end Scienti c Applications on Commodity Microprocessors." In ISPASS, 2008.

[MPA16] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. \Tabla: A uni ed template-based framework for accelerating statistical machine learning." In HPCA-22, 2016.

[MSB05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. \Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset." ACM SIGARCH Computer Architecture News, 2005.

[MWN15] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. \SNNAP: Approximate computing on programmable socs via neural acceleration." In HPCA-21, 2015.

[NHW07] Umesh Nawathe, Mahmudul Hassan, Lynn Warriner, King Yen, Bharat Upputuri, David Greenhill, Ashok Kumar, and Heechoul Park. \An 8-core, 64-thread, 64-bit, power e cient SPARC SoC (Niagara 2)." ISSCC, 2007.

[NM13] Stephen Neuendor er and Fernando Martinez-Vallina. \Building zynq accelerators with Vivado high level synthesis." In FPGA, 2013.

[OPH15] Lena E Olson, Jason Power, Mark D Hill, and David A Wood. \Border control: sandboxing accelerators." In MICRO-48, 2015.

[ORK15] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. \Toward accelerating deep learning at scale using specialized hardware in the datacenter." In Hot Chips, 2015.

[PAC97] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. \A case for intelligent RAM." IEEE micro, 17(2):34{44, 1997.

[PCC14] Andrew Putnam, Adrian M Caul eld, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. \A recon gurable fabric for accelerating large-scale datacenter services." In ISCA-41, 2014.

[PHB14] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. \Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Uni ed Address Spaces." In ASPLOS-XIX, 2014.

[PHW14] Jonathan Power, Mark D Hill, and David A Wood. \Supporting x86-64 address translation for 100s of GPU lanes." In HPCA-20, 2014.

[PPM09] Hyunchul Park, Yongjun Park, and Scott Mahlke. \Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications." In MICRO-42, 2009.

[PRM17] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. \Scnn: An accelerator for compressed-sparse convolutional neural networks." In ISCA-44, 2017.

[QHS13] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. “Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing.” In *ISCA-40*, 2013.

[RAS14] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. “CNN features off-the-shelf: an astounding baseline for recognition.” In *CVPR*, 2014.

[rob18] “The Mobile Robot Programming Toolkit.” https://www.mrpt.org/, 2018.

[RWA16] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. “Minerva: Enabling Low-power, Highly-accurate Deep Neural Network Accelerators.” In *ISCA-43*, 2016.

[SBJ15] J Stuecheli, Bart Blaner, CR Johns, and MS Siegel. “Capi: A coherent accelerator processor interface.” *IBM Journal of Research and Development*, **59**(1):7–1, 2015.

[SBS16] Sagi Shahar, Shai Bergman, and Mark Silberstein. “ActivePointers: a case for software address translation on GPUs.” In *ISCA-43*, 2016.

[SCS08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. “Larrabee: a many-core x86 architecture for visual computing.” *ACM Transactions on Graphics (TOG)*, **27**(3):18, 2008.

[sda18] “SDAccel Development Environment.” https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html, 2018.

[SDS00] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. “Recency-based TLB Preloading.” In *ISCA-27*, 2000.

[SGB14] Sudharsan Seshadri, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. “Willow: A User-Programmable SSD.” In *OSDI*, 2014.

[SGG12] Manish Shah, Robert Golla, Gregory Grohoski, Paul Jordan, Jama Barreh, Jeffrey Brooks, Mark Greenberg, Gideon Levinsky, Mark Luttrell, Christopher Olson, Zeid Samoail, Matt Smittle, and Thomas Ziaja. “Sparc T4: A Dynamically Threaded Server-on-a-Chip.” *IEEE Micro*, **32**(2):8–19, March 2012.

[SL11] Malcolm Singh and Ben Leonhardi. “Introduction to the IBM Netezza warehouse appliance.” In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 385–386. IBM Corp., 2011.

[SNM16] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. “ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars.” In *ISCA-43*, 2016.

[SPM16] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. "From high-level deep neural models to FPGAs." In *MICRO-49*, 2016.

[SQL17] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. "PipeLayer: A pipelined ReRAM-based accelerator for deep learning." In *HPCA-23*, 2017.

[SRH17] Bharat Sukhwani, Thomas Roewer, Charles L Haymes, Kyu-Hyoun Kim, Adam J McPadden, Daniel M Dreps, Dean Sanner, Jan Van Lunteren, and Sameh Asaad. "Contutto: a novel FPGA-based prototyping platform enabling innovation in the memory subsystem of a server class processor." In *MICRO-50*, 2017.

[SRW15] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. "The aladdin approach to accelerator design and modeling." *IEEE Micro*, **35**(3):58–70, May 2015.

[SSB16] Yakun Sophia Shao, Sam Likun Xi Vijayalakshmi Srinivasan, and Gu-Yeon Wei David Brooks. "Co-designing accelerators and soc interfaces using gem5-aladdin." In *MICRO-49*, 2016.

[SXS15] Yakun Sophia Shao, Sam Xi, Viji Srinivasan, Gu-Yeon Wei, and David Brooks. "Toward Cache-Friendly Hardware Accelerators." In *Sensors to Cloud Architectures Workshop (SCAW), in conjuction with HPCA-21*, 2015.

[SYW13] Richard Sampson, Ming Yang, Siyuan Wei, Chaitali Chakrabarti, and Thomas F. Wenisch. "Sonic Millip3De: A Massively Parallel 3D-stacked Accelerator for 3D Ultrasound." In *HPCA-19*, 2013.

[SZ14] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556*, 2014.

[TH94] Madhusudhan Talluri and Mark D. Hill. "Surpassing the TLB Performance of Superpages with Less Operating System Support." In *ASPLOS-VI*, 1994.

[VHR08] Sriram R Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, et al. "An 80-tile sub-100-w teraflops processor in 65-nm cmos." *IEEE Journal of Solid-State Circuits*, **43**(1):29–41, January 2008.

[VSG10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. "Conservation Cores: Reducing the Energy of Mature Computations." In *ASPLOS-XV*, 2010.

[WGH07] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. "On-Chip Interconnection Architecture of the Tile Processor." *IEEE Micro*, **27**(5):15–31, September 2007.

[WIA14] Louis Woods, Zsolt István, and Gustavo Alonso. "Ibex: an intelligent storage engine with support for advanced SQL offloading." *Proceedings of the VLDB Endowment*, **7**(11):963–974, 2014.

[WL03] Daniel Wiklund and Dake Liu. "SoCBUS: Switched Network on Chip for Hard Real Time Embedded Systems." In *IPDPS*, 2003.

[WM95] Wm A Wulf and Sally A McKee. "Hitting the memory wall: implications of the obvious." *ACM SIGARCH computer architecture news*, **23**(1):20–24, 1995.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces." In *VLDB*, 1998.

[WSR05] Pascal T. Wolkotte, Gerard J. M. Smit, Gerard K. Rauwerda, and Lodewijk T. Smit. "An Energy-Efficient Reconfigurable Circuit-Switched Network-on-Chip." In *IPDPS*, 2005.

[Xil17] Xilinx. *ADM-PCIE-7V3 Datasheet*, 1 2017. Rev. 1.3.

[XLJ16] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, et al. "Bluecache: A scalable distributed flash-based key-value store." *Proceedings of the VLDB Endowment*, **10**(4):301–312, 2016.

[YYD16] Salessawi Ferede Yitbarek, Tao Yang, Reetuparna Das, and Todd Austin. "Exploring specialized near-memory processing for data intensive operations." In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 1449–1452. IEEE, 2016.

[YZS14] Jieming Yin, Pingqiang Zhou, Sachin S. Sapatnekar, and Antonia Zhai. "Energy-Efficient Time-Division Multiplexed Hybrid-Switched NoC for Heterogeneous Multicore Systems." In *IPDPS*, 2014.

[ZDZ16] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. "Cambricon-X: An accelerator for sparse neural networks." In *MICRO-49*, 2016.

[ZFZ16] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks." In *ICCAD*, 2016.

[ZLS15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. "Optimizing FPGA-based accelerator design for deep convolutional neural networks." In *FPGA*, 2015.