OVERCOMING THE LIMITATIONS OF ACCELERATOR-CENTRIC ARCHITECTURES WITH MEMOIZATION-DRIVEN SPECIALIZATION

ADI FUCHS

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING
ADVISER: PROFESSOR DAVID WENTZLAFF

JUNE 2019
Abstract

Modern computer systems are undergoing a paradigm shift towards specialized architectures that improve computation efficiency by orders of magnitude. At the core of specialized architectures are “accelerator” chips that are designed to target specific computation domains and achieve better gains under a given hardware budget. Accelerator-centric chip specialization has become one of the leading approaches to mitigate the gap between the growing computation demands and the dwindling improvement in CMOS chips’ budget and capabilities imposed by the end of Moore’s law and failure of Dennard scaling.

The accelerator approach suffers from several inherent caveats. (i) While domain-specific accelerators improve the gains of a non-improving CMOS budget, they are implemented using CMOS transistors. Therefore, accelerator chips are susceptible to the end of CMOS scaling. Much like general-purpose chips, their gains will be capped following the end of CMOS scaling, and: (ii) the targeting of specific domains requires the sacrificing of programmability, which makes ASIC accelerators limited in their ability to efficiently accommodate new applications and features, due to the long process of ASIC accelerator development.

In this dissertation, we explore the limits of accelerators imposed by the combination of transistor-dependence and diminishing specialization returns. We develop architectures that employ memoization to extend the reach of traditional hardware specialization by alleviating transistor-dependence and loss-of-programmability in accelerator-centric architectures.
Acknowledgments

I thank my family, friends, collaborators, and coordinators that have helped me through this intellectually-demanding, emotionally-turbulent, and highly-rewarding endeavor. Your contributions and support were priceless during these times, and I will cherish our times together.

First and foremost, I would like to thank my advisor, David Wentzlaff, for countless hours brainstorming, mentoring, thinking about the big problems, coming up with great ideas, and balancing a long-term vision with a well-grounded approach towards research. It was an honor and a privilege collaborating with a person as smart, as inspiring, and as excited about science and engineering as yourself. I thank you for your great confidence and your belief in me during the early (and unexpectedly turbulent) times at Princeton. You have inspired me to always strive to be a better researcher, and explore new and unknown fields, away from my comfort zones. Your curiosity, persistence, and optimism kept me going while facing the many challenges entailed in this long journey.

At Technion, I was fortunate to have great minds and mentors as my Master’s advisors, Yoav Etsion and Uri Weiser. I thank you for being my first true mentors. Your guidance has provided me with the tools and opportunities I needed to become a capable researcher. The early lessons you have taught me have resonated through the various stage of my research, and have helped me in rapidly bringing up and testing new ideas. I thank my collaborators Avi Mendelson, Shie Mannor, and Noam Shalev, that along with my advisors, helped me in getting my first research works accepted.

I am very thankful to have world-class researchers serving as members of my dissertation committee: Niraj Jha, Ruby Lee, Margaret Martonosi, and Prateek Mittal. I thank them for their assistance, time, and useful advice in improving the quality of my research papers and this thesis.
I thank Ruby Lee for being an inspiring collaborator that helped me in channeling my prior knowledge and experiences in getting my first published research paper at Princeton, and for serving on my dissertation committee.

I thank my inspringly bright and eager lab mates at the Princeton parallel architecture group: Yanqi, Yaosheng, Tri, Mike, Zhuozhi, Jon, Alexey, Mohammad, Ting-Jung, Paul, Ang, Georgios, and Fei. I also thank my lab mates and friends at the Technion Computer Engineering Center: Dima, Yuval, Shahar, Tomer, Elad, Hanna, Deborah, and Ruth. Thank you all for being there, for your kind advice, intriguing exchange of thoughts, and keeping the morale high.

I am grateful for the administrative staff in the Electrical Engineering department: Lori, Colleen, Roeli, Stacey, and Katie. I thank you for supporting and for tending to all my needs during my years at Princeton.

This work was supported in part by the NSF and DARPA.

I am forever thankful for my parents, Anat and Miki, their life partners, Yosi and Vered, and my sister Moran, for your love and kindness that have shaped me to be the person that I am today. I thank my wife’s family for being there for us and for supporting us. Last but not least, my wife and fellow graduate student, Anat, who is my family and best friend through the good times and the bad, for being my full partner in raising our daughters Noga and Ella, our crown jewels that always shine at the end of long and intensive research days. You have kept me going and helped me see the truly important things in life. I love all of you.
To my wife, Anat, and our daughters Noga, and Ella, and to my parents and sister.
## Contents

Abstract ................................................................. iii  
Acknowledgments ....................................................... iv  
List of Tables ............................................................ xi  
List of Figures ........................................................... xiii  

1 Introduction ......................................................... 1  
   1.1 The Rise of Accelerator-Centric Architectures ................. 1  
   1.2 Accelerator Limitations ........................................... 3  
   1.3 Overcoming Accelerator Limitations with Memoization ......... 3  
   1.4 Thesis Organization and Sources .................................. 5  
   1.5 Previous Work ....................................................... 5  

2 Background and Related Work ..................................... 7  
   2.1 Transistor Scaling .................................................. 7  
       2.1.1 Early 2000s: The Power Wall, Rise of Multicore Architectures .... 8  
       2.1.2 Mid 2000s: The Utilization Wall (a.k.a. Dark Silicon), Rise of Accelerators ........................................... 9  
       2.1.3 2020s: The Accelerator Wall, What’s Next? .................... 9  
   2.2 Programmability of Accelerators (or Lack of) .................. 10  
   2.3 Related Work ........................................................ 12  
       2.3.1 Accelerators and CMOS Limitations ............................ 12
3 The Accelerator Wall

3.1 Introduction ......................................................... 19
3.2 The Interplay of CMOS Scaling and Chip Specialization Return .......... 22
3.3 CMOS Potential Model ............................................. 24
3.4 Empirical Specialization Returns ................................ 27
  3.4.1 ASIC Video Decoders .......................................... 27
  3.4.2 GPU Graphics Rendering ...................................... 29
  3.4.3 FPGA Convolutional Neural Networks ......................... 32
  3.4.4 CPU/GPU/FPGA/ASIC Bitcoin Miners ....................... 34
  3.4.5 Observations and Insights .................................. 35
3.5 Chip Specialization: Concepts .................................... 37
  3.5.1 Chip Specialization Concepts ................................ 37
  3.5.2 The Limits of Chip Specialization Concepts ................... 39
3.6 Gains of Chip Specialization Concepts ................................ 43
3.7 The Accelerator Wall .............................................. 46
3.8 Conclusions .......................................................... 51

4 COREx: A Compute-Reuse Architecture for Datacenter Accelerators ... 52

4.1 Introduction ......................................................... 53
4.2 Background and Motivation ....................................... 56
  4.2.1 Specialization and Scalability ................................ 56
  4.2.2 Memoization Requirements: Three C’s .......................... 57
  4.2.3 Reuse Opportunities in Datacenters ............................ 59
4.3 The COREx Architecture ........................................... 62
5 REDS: A REuse-Driven Specialization Architecture for Accelerator-less Clouds

5.1 Introduction ................................................. 84

5.2 Background and Motivation ................................. 87

  5.2.1 A Shift in Cloud Computing Paradigms .......... 87

  5.2.2 Accelerator-less Cloud Domains ................. 88

  5.2.3 A Primer on Reuse-Driven Specialization ...... 90

5.3 Reuse-Driven Specialization: Opportunities and Challenges .... 93

  5.3.1 Cloud Power Laws .................................. 93

  5.3.2 Caching Computations at Cloud Scale .......... 93

  5.3.3 Filtering and Commonality at Cloud Scale ...... 95

5.4 Architecture .................................................. 99

  5.4.1 Software Primitives and API ...................... 100

  5.4.2 Design ............................................... 102

  5.4.3 REDS Flows .......................................... 104

  5.4.4 End-to-End Flow .................................. 105

5.5 Evaluation ..................................................... 109
List of Tables

2.1 Taxonomy of Related Datacenter Acceleration, Memoization, and NVM Accelerator Studies. .................................................. 13
2.2 Properties and Scopes of Related Studies. .............................. 17
3.1 Chip Specialization Concepts. Examples From a TPU ASIC Chip. . . . 39
3.2 Summary of Time and Space Complexity Limits for Chip Specialization Concepts, in Terms of DFG Definitions. ......................... 43
3.3 CMOS-Specialization Sweep Parameters. .............................. 44
3.4 Evaluated Applications and Domains. ................................. 45
3.5 Accelerator Wall: Physical Parameters. ............................... 47
4.1 Accelerator Design Sweep Parameters. .................................. 71
4.2 Evaluated Workloads, Input Sources, and Use-Cases. ............... 75
4.3 Kernels and Optimal Design Sweep Points. ............................ 77
4.4 General System Parameters. ............................................ 77
4.5 Optimal Table Configurations for a Subset of Workloads, Following Memory-Layer Specialization; CHT Total Size and Layout ($N \times M = N$ Interleaved $\times M$ Parallel Banks), CHT Bank Latency, and ILU Size. ILUs are 4-Way Set-Associative. .................................................. 78
5.1 Qualitative Comparison: GPPs, Programmable Accelerators (e.g., FP-GAs), DSAs, and GPPs with Reuse-Driven Specialization. ............ 87
5.2 Impact of Optimization Techniques. ........................................ 99
5.3 Accelerator Design Sweep Parameters. .............................. 107
5.4 Compute Cluster Parameters ............................................... 112
5.5 Evaluated Functions. Input and Output Sizes, Runtimes and Energies for
    Fixed-Function Accelerators and CPUs. ............................... 112
5.6 REDS Design Sweep Parameters. ....................................... 113
List of Figures

1.1 Main Drivers of the Accelerator Revolution. ........................................ 2

2.1 Processor Frequency and Core Count Versus Manufacturing Dates. .... 8

2.2 Qualitative Comparison between Accelerators and a General-Purpose Pro-
cessor. ........................................................................................................... 11

3.1 Evolution of Bitcoin Mining ASIC Chips. Performance Metric is SHA256
Hashing Throughput per Chip Area (Hashes/Seconds/mm^2). ................. 21

3.2 Abstraction Layers: Traditional and Accelerated Systems. Dashed Box
Groups the Layers of Specialization. ............................................................ 23

3.3 CMOS Potential Model: Device Scaling, Chip Transistor Budget, and
Physical Chip Gains. .................................................................................... 25

3.4 Video Decoder ASICs: Performance, Hardware Budget, and Energy Effi-

3.5 GPU Frame Rates (Apps 1-5): Throughput and Energy Efficiency .... 30

3.6 Architecture + CMOS Scaling: Throughput ......................................... 31

3.7 Architecture + CMOS Scaling: Energy Efficiency ................................. 31

3.8 FPGA Implementations of AlexNet and VGG-16. Sources: FPGA2016 [164],
FPGA2016† [139], FPGA2016* [112], FPGA2017 [164], FPGA2017† [200],
FPGA2017* [20], FPGA2018 [155], FPGA2018† [197]. ................................. 33
3.9 Bitcoin Mining Capabilities of CPU, GPU, FPGA and ASIC chips (vs. AMD Athlon 64 CPU Miner). 36
3.10 TPU: DNN Inference + Specialization Concepts 39
3.11 DFG Example: 3 Inputs, 2 Computation Stages, 2 Outputs (An Example Computation Path is Marked in Red). 41
3.12 Visualization of a 3D Stencil Computation 46
3.13 3D Stencil Power, Timing, and CMOS Sweep. Arrows Highlight Optimal Point and Gain Sources. 46
3.14 Specialization and CMOS Accelerator Gains. 47
3.15 Evaluated Applications and Specialization Chips: Accelerator Performance Projections. 49
3.16 Evaluated Applications and Specialization Chips: Accelerator Energy Efficiency Projections. 50
4.1 Accelerator with Compute-Reuse Storage. 55
4.2 Transistor, Accelerator, and NVM Trends. Marked Points are Existing Systems; Dashed Lines are Projections. 57
4.3 Traffic Reuse and Storage Requirements across 100K Web Searches; A Retrieved versus Reused Records, B Retrieved versus Uniquely Stored Traffic Size. 60
4.4 Video Encoding Execution Time Breakdown. 61
4.5 Macroblock Reuse Granularity in Video Frames. Reused Blocks Are Marked in White. 63
4.6 Accelerator Execution Flavors. 65
4.7 COREx Top-level Block Diagram. 66
4.8 The COREx Pipeline. 66
4.9 COREx Execution across the Different COREx Components. 67
4.10 Motion Vector Construction; ① the Previous Frame. ② Current Frame ③ the Constructed Motion Vector. ........................................ 69

4.11 SAD Design Sweep across Traditional (“BASE”) and Processing-In-Memory (“PIM”) Accelerators. ........................................ 72

4.12 SAD Trace Table Exploration. Hit, Miss, and False-Hit Rates as a Function of ILU+CHT Space Sizes. ........................................ 73

4.13 COREx Gain Space Exploration for SAD. Optimal Gains and Design Points Are Detailed in Each Graph. ........................................ 74

4.14 COREx Energetic and Runtime Gains. ........................................ 79

4.15 Energetic Breakdown for Energy-OPT Schemes. ...................... 81

5.1 Specialization in a Cloud Compute Cluster. Domain-Specific Acceleration for Mature Domains. Reuse-Driven Specialization for “Accelerator-less” Domains. ........................................ 86

5.2 U.S. Smart Speaker Functionality Trends. .................................... 90

5.3 Runtime-Energy Space for a K-nearest neighbors function. Fixed-Function Accelerator versus a CPU with Reuse-Memory, for Different Reuse Hit Rates. ........................................ 91

5.4 Power Laws in Commercial Cloud Services. ......................... 94

5.5 Key Reuse Distance for Different Distributions. ....................... 96

5.6 Impact of Filtering on Miss Rate and Pollution. ....................... 97

5.7 Miss Distribution with Respect to Key Rank, for Different Traffic Skews (α Values) and Sharing Degrees. .............................. 99

5.8 Impact of Domain Volatility and Commonality for α = 1.6. New Epochs Signify Computation Updates. Computations Cannot Be Reused In Different Epochs. .............................. 100

5.9 Top-Level REDS Architecture. ............................................... 102

5.10 Microarchitectural Diagram of the REDS Modules. ................... 104
5.11 Pipeline Diagrams of Hardware REDS Flows. Colors Signify Different Components. ................................................. 105
5.12 SPMV-Ellpack Fixed-Function Accelerator Exploration. Design Sweep + Optimal Points. ................................................ 107
5.13 Impacts of Traffic Bias and Congestion. ..................................................... 110
5.14 REDS Design Exploration Flow ................................................................. 111
5.15 REDS Design Space Exploration of the SPMV Kernel. Trace Analysis, and Timing and Energy Sweeps. ........................................ 111
5.16 Speedups for Runtime-OPT REDS (vs. Accelerators). Higher is Better. . 114
5.17 Relative Energies for Energy-OPT REDS (vs. Accelerators). Lower is Better. 114
5.18 Runtime Breakdowns for Runtime-OPT REDS. ........................................... 115
5.19 Energetic Breakdowns for Energy-OPT REDS. .......................................... 115
Chapter 1

Introduction

An accelerator is a processing chip, or a unit within a processing chip, that is designed to execute specific applications or computation domains efficiently. By targeting specific computation patterns, accelerators improve silicon performance under a given cost (e.g., power, energy, area, or money) by orders-of-magnitude, compared to traditional, fully-programmable, general-purpose processors (such as x86-based Intel processors). Accelerators have become prominent building-blocks for contemporary architectures, and their adoption is expected to rise as computation demands increase.

1.1 The Rise of Accelerator-Centric Architectures

In recent years, the world of computing underwent major changes that have resulted in a golden-age for accelerators. In specific, and as shown in Figure 1.1, we attribute the popularity of accelerators to three major changes:

- **Applications**: The growing popularity of domains such as deep learning [51] and graph processing that produce classes of “killer apps” for specialized chips. These are compute-intensive applications with high degrees of parallelism, simple control-flows, and exhibit heavy reuse of well-defined computation patterns. These prop-
Figure 1.1: Main Drivers of the Accelerator Revolution.

Properties were shown to make applications such as deep-learning or video decoding convenient targets for acceleration [127].

- **Platforms**: The growing usage of platforms in which efficiency is critical. The majority of the world’s computational payload is shifting towards battery-limited mobile devices such as smartphones [141], or power-hungry datacenters, that are projected to consume energy at an annual rate of 73 billion kWh by 2020 in the U.S. alone [154].

- **Transistors**: The slowdown in CMOS transistor scaling that produces high rates of chip power [56,178]. CMOS scaling is currently projected to completely end at around 2021 [4], following which transistors would cease to improve. While the “applications” and “platforms” trends made accelerators popular and appealing, CMOS-imposed limitations made accelerators an almost necessity. At this point, researchers, architects, and designers acknowledged that *it would become impossible for us to keep improving our computer systems, without changing the way we design processors.*
1.2 Accelerator Limitations

To combat the end of Moore’s Law, researchers in industry and academia employ accelerators to improve application gains using non-improving chip transistor budgets. The core premise of accelerator development is that, by compromising on programmability to a certain degree and focusing on domain-specific computation patterns, it is possible to devise efficient computation engines in hardware and reduce hardware/software interfaces to support the mission-critical functionally.

While gains from accelerators can be significant, there are two main caveats to the accelerator approach. (i) Commercial accelerators, such as GPUs, FPGAs, or ASICs, are CMOS-based chips that are implemented using transistors and are therefore transistor-dependent by definition. Since there is a finite number of ways to map a computational problem to hardware under a fixed budget, the chip-based specialization returns of applications or domains are conceptually limited. Once transistors stop improving and hardware budgets plateau, accelerators will encounter diminishing returns, and reach a limit which we term as “the accelerator wall.” (ii) To achieve high-efficiency, architects and designers are required to redefine the programming stack and develop efficient, ad-hoc, hardware chips (e.g., ASICs). These processes require long development and deployment times and are unfit to a rapidly-changing environment such as future function-centric cloud services and datacenters.

1.3 Overcoming Accelerator Limitations with Memoization

“Memoization” is a computation optimization technique first suggested by Donald Michie in 1968 [118]. In memoization, tables store the results of previous computations. Once the computing system detects the recurrence of a previously computed problem, the cor-
responding computation result is fetched from the tables instead of being redundantly re-computed.

We are proposing memoization as an extension to contemporary chip-based specialization, to overcome the technological and economic limitations of specialized accelerator chips. The rationale for this proposal stems from multiple factors. (i) In contrast to accelerator-based specialization, memoization is driven by table storage (i.e., memory size). As we show, the density of emerging memories such as Resistive RAMs (ReRAMs), Spin-Torque Transfer RAMs (STT-RAMs), and Phase-Change Memories (PCMs) is projected to scale after transistor scaling stops, therefore memoization-specialization using emerging memories has a better trajectory than chip specialization that will become limited by the end of transistor scaling. (ii) Conceptually, memoization extends the reach of existing specialization. If chip-based specialization relies on knowing specific characteristics of the problem, memoization relies on knowing the problem and the solution. By exploring different table sizes and layouts, it is possible to devise an application or workload specific memory hierarchy that improves overall performance or energy efficiency. (iii) In contrast to accelerator-based specialization, memoization is microarchitecturally-agnostic. It does not rely on specific application characteristics (e.g., degrees of memory level parallelism or domain-specific hardware units to carry out non-trivial operands such as tangent calculations for some deep-learning architectures). It only relies on the most basic invariants of the computation; the fact that the computation has an input and an output. Therefore, with a programmable controller, it is possible to adapt to workload or functionality changes and use memoization as a programmable form of specialization that extends general-purpose cores and avoid changes in the hardware/software stack or long accelerator development and deployment times.
1.4 Thesis Organization and Sources

The rest of the thesis is organized as follows. Chapter 2 overviews the history and related studies of architectural limitations and the employment of memoization and non-volatile memories. In Chapter 3, we present the accelerator wall, a study introducing methods and tools for accelerator evaluation while projecting the limits of popular chip-based specialized applications at the end of Moore’s Law. Chapter 3 is based on our works entitled: “The Interplay of Transistors and Accelerators” [62] and: “The Accelerator Wall: Limits of Chip Specialization” [64]. In Chapter 4, we present a memoization-driven architecture that leverages still-scaling emerging memories to extend chip-based specialization for accelerators beyond the end of Moore’s Law. In Chapter 5, we present an architecture that employs memoization to provide specialization for accelerator-less clouds. Finally, in Chapter 6 we conclude our findings and this dissertation.

Chapters 2+3 are based on our works entitled: “The Interplay of Transistors and Accelerators” [62] and: “The Accelerator Wall: Limits of Chip Specialization” [64]. Chapters 2+4 are based on our work entitled: “Scaling Datacenter Accelerators with Compute-Reuse Architectures” [63].

1.5 Previous Work

Portions of this thesis appeared in the following:


Chapter 2

Background and Related Work

2.1 Transistor Scaling

Transistors have been the driving force behind the evolution of computer systems, and the semiconductor industry in specific, for over fifty years. While investigating manufacturing costs and the complexity of functions integrated onto processing chips by the late 1950s, Gordon Moore was able to detect industry trends and established: **“Moore’s Law”** in 1965 [122]. Moore’s Law projected that the number of transistors that are integrated on a chip would double about every two years (the original projection was about a year). In a later study, published in 1974, Robert Dennard analyzed metal-oxide-semiconductor (MOS) transistors and showed that when scaling their voltage, transistors not only get smaller, but they can become twice as fast, under the same fixed power density. The scaling behavior of MOS transistors was commonly known as **“Dennard Scaling”** [54]. When combining both rules, the computer industry has evolved at a staggering rate; when comparing a 1970s 4004 processor and a 2015 6th generation Core processor, transistor dimensions shrunk by $700 \times$, performance improved by $3,400 \times$, energy efficiency improved by $90,000 \times$, and the cost per transistor went down by $60,000 \times$ [80]. But in recent years,
transistor-imposed limitations have impeded processor evolution multiple times, each time they posed a different wall that was resolved by changing the way processors were built.

2.1.1 Early 2000s: The Power Wall, Rise of Multicore Architectures

In the period between the 1970s until early 2000s, processors relied on CMOS scaling to achieve faster circuitry that enabled deep pipelines, complex microarchitectures, and higher core clock frequency. The increase in clock frequency resulted in high power density rates and induced processor temperatures that reached the dissipation limits of conventional processor cooling systems [135]. As shown in Figure 2.1a, the power wall resulted in the plateauing of processor frequencies at around 2004. To meet the growing computation demands of applications, processor manufacturers decided that since transistors cannot go faster, but budgets are still increasing due to transistor scaling, the proper solution would be pivoting from the fast single-core architecture model, to a model that integrated multiple cores running at lower frequencies. As Figure 2.1b shows, starting in 2005 core count continuously increased, and single-core processors were no longer manufactured starting 2014.
2.1.2 Mid 2000s: The Utilization Wall (a.k.a. Dark Silicon), Rise of Accelerators

Starting in the early 2000s, the increase in the number of on-chip processing cores became the prime means for harvesting the exponential returns entailed by the increase in the number of on-die transistors. Unfortunately, the trend of increasing of cores turned out to be infeasible by about mid-2000s, following the breakdown of Dennard scaling. Transistor voltage, and consequently transistor leakage power, stopped scaling at the same rate as transistor density [178]. Therefore, while transistors were still shrinking and we could physically fit an exponentially growing number of transistors across CMOS generations, it became impossible to employ them since keeping more transistors active means using more power. Therefore, packaging limitations that dictate strict thermal power budgets (TDP) caused an exponential drop in die utilization, leading to a phenomenon known as the “Utilization Wall” or “Dark Silicon”, and it capped the number of cores that can be used by multicore architectures, even under lower clock frequencies [56]. The limitations imposed by dark silicon caused the rise of specialized chips, also known as “accelerators”. The idea was to compromise on programmability in favor of chip specialized circuity and designs, and through silicon customization, we can create accelerator-based architectures that efficiently employ only the mission-critical transistors for a given computation [170].

2.1.3 2020s: The Accelerator Wall, What’s Next?

In contrast to previous scaling limitations, current projections predict the definite ending of transistor scaling at around 2021, in what is commonly known as the end of Moore’s Law. While it is well-accepted that this would end the evolution of general-purpose processors, in Chapter 3, we predict that the end of Moore’s Law will end the evolution of domain-specific accelerators as well, in what we term as the accelerator wall. Paraphras-
ing Aeschylus’s Agamemnon play in 458 BCE: “accelerator chips live by transistors, and die by transistors.”

Since the density of emerging non-volatile memories is projected to scale for about another decade [3,4], in Chapter 4, we leverage still-scaling non-volatile memories to develop a memoization-driven specialization layer to extend the gains of accelerators beyond the end of Moore’s Law. This is done by shifting computations from the non-scaling transistor domain to the still-scaling memory domain.

2.2 Programmability of Accelerators (or Lack of)

The targeting of specific domains requires accelerator developers to sacrifice some degree of programmability. Since most accelerators are programmable to some extent, there is no clear or formal distinction between a general-purpose processing chip and an accelerator chip. An accelerator can still run a wide range of applications (and even all possible applications). However, compared to general-purpose processors, accelerators have a higher disparity when it comes to application execution. Figure 2.2 highlights the differences between a general-purpose processor and accelerators via a qualitative view of the attained performance per cost (y-axis) with respect to the application space (x-axis). This qualitative model is inspired by Yavits et al. [194]. As the figure shows, accelerators will likely perform poorly when they are executing domains that are not in their targeted scope. For example, it does not make sense to run single-threaded applications with many complex control flows on a GPU, as they will cause highly underutilized hardware resources, and the GPU will end up performing worse than a single-core general-purpose CPU with a high-end branch predictor.

While the gains from domain specialization through compromised programmability are significant, this entails several costs. Aside from incurring non-recurring engineering costs [96] and the fact that loss of programmability can impede algorithmic innova-
Figure 2.2: Qualitative Comparison between Accelerators and a General-Purpose Processor.

accelerators require a long deployment time. For example, in 2013, following the projections of computational demands for speech recognition, Google decided to develop the Tensor Processing Unit (TPU) ASIC chips. The first TPUs were developed, verified, and deployed in datacenters over a period of 15 months [88]. Similarly, in 2016, Microsoft began developing the FPGA-based DNN cloud acceleration architecture under the code name “project Brainwave [60]” for FPGA-accelerated DNNs in the cloud. About a year later, according to the Azure blog, the first boards of the Brainwave neural processing unit (NPU) were deployed [120]. In both cases, it took a year from initial identification of computation need to the deployment of first accelerators. During this period, DNN cloud acceleration remained accelerator-less, and was dependent solely on inefficient general-purpose processors. Therefore, long deployment times make accelerators like the TPU and the NPU impractical for cloud platforms handling emerging domains or domains with rapidly-changing functionality. In Chapter 5, we tackle the problem of long accelerator deployment times by integrating memoization-driven layers with fully-programmable processors to achieve near-accelerator efficiency rates, while not compromising on programmability.
2.3 Related Work

In this section, we discuss prior studies with relevance to the engaged topics in Chapter 3, Chapter 4, and Chapter 5.

2.3.1 Accelerators and CMOS Limitations

Several works have explored the limiting factors of processing technology and innovation. One of the earliest works tried to project the practical limits of instruction-level parallelism [183]. Recent studies identified the dark silicon phenomenon that caps the number of active chip transistors [56,178]. These studies demonstrated how the slowdown of CMOS scaling and power budget limitations in multicore chips would necessitate a shift towards new architectures. In a later study, Taylor advocated for specialization as one of four avenues to mitigate dark silicon [170], and it was also motivated by Halpern et al. [74] who examined trends in mobile and desktop processors. While specialized chips are motivated by the slowdown of CMOS scaling, we demonstrate that they are prone to transistor-imposed limitations as well.

Recent studies on specialized hardware improve programmability to maintain hardware performance and relevance amid frequent algorithmic changes [126], optimize while accounting for non-recurring engineering costs in ASIC development [96], and use DRAMs to replace inefficient SRAMs in FPGAs [66]. Accelerator modeling expressed accelerated applications as transformable dependence graphs [128] and dynamic dependence graphs in Aladdin [151].

A study by Borkar and Chien decoupled chip and application performance to estimate the impacts of microarchitecture on general-purpose microprocessors [32]. In recent studies, the nearing-end of single monolithic chips was the motivation for devising multi-chip GPU architectures [18] and emerging memory based memoization architectures for ac-
Table 2.1: Taxonomy of Related Datacenter Acceleration, Memoization, and NVM Accelerator Studies.

<table>
<thead>
<tr>
<th>Memoization</th>
<th>Accelerators/Big-Data or Datacenter Apps</th>
<th>Emerging Memories</th>
<th>Massive Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[11,42,45,81,131,142,157,165,185]</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>[9]</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>[17,146,167]</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>[84]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[35,73,144]</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>[37,77,106,113,137]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[47,82,89]</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>[158,184,198]</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>[31,38,149,195]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[126–128]</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
</tbody>
</table>

To the best of our knowledge, our work is the first to quantify transistor dependence in accelerators and explore the limitations of chip specialization foundations.

### 2.3.2 Datacenter Acceleration, Memoization, Programmable Acceleration, and NVM Accelerators

Table 2.1 summarizes the different properties of previous studies and how they relate to the ideas explored in Chapter 4. To the best of our knowledge, no previous hardware memoization study has explored coarse-grained memoization for accelerators nor the use of large NVM-based tables for memoization, no previous software memoization study has used memoization for accelerated applications, and no study has suggested memoization as a tool to alleviate the transistor-based scalability limitation in accelerated systems.

**Datacenter Acceleration**: The appealing performance efficiency of hardware specialization and recent ubiquity of ASICs and FPGAs motivated the integration of specialized hardware in datacenters and cloud servers. Academic studies developed specialized hardware for datacenter applications such as Memcached [106], Apache and Hadoop [82]. Planet-scale ASIC clouds were proposed for Bitcoin mining and video transcoding [113], and DRAM-based reconfigurable datacenter fabrics for the effects of “dark-silicon” in
servers [77]. In industry, commercial datacenters have begun shifting towards heterogeneity with recent efforts of specialization using FPGAs [37,137], ASICs [89], and GPUs [47]. As shown in this work, we believe these schemes and future accelerated datacenters would benefit from memoization, as it would increase their computation scalability using emerging memories.

**Programmable Acceleration:** Prior works identified basic computation patterns that dominated the behavior of accelerated workloads. They also studied how the workloads affect the way accelerator architectures are built (e.g., data reuse, concurrency, etc.). Using these insights, these works aimed at improving the scope of accelerators beyond domain specialization [126,127]. In addition, prior work strived to improve the efficiency of general-purpose processors using specialized programmable cores [128].

**Memoization:** Memoization has been extensively explored in past work and is supported by modern programming environments. To the best of our knowledge, it has not yet been explored as a means to trade CMOS-based computations for scalable memory technologies, nor has it been applied to entire kernels in accelerated systems.

Hardware memoization schemes were studied prior to the emergence of accelerators. In contrast to COREx, they were implemented using small tables and targeted fine-grained recurring computations such as instructions [42,157], math functions [142], floating-point operations [11], instruction scheduling sequences [131], and basic blocks [81].

Compiler and code profiling-centric memoization studies employ optimizations to memoize different targets by exploiting redundancy in distinct code regions, such as loops [45,185], pure functions [165], and shared library code for datacenter applications (e.g., Memcached [9]). Recent efforts to integrate memoization into datacenters focus on changes to the software stack to improve storage [73], or eliminate redundant computations via iterative processing in MapReduce frameworks [35,144].

Compared to existing software and datacenter memoization, COREx is fit for accelerated architectures. Its flow does not require any specific workload assumptions or program-
ming framework properties to guarantee memoization correctness. COREx gains from the low runtime overhead of hardware implementation. For example, the typical latencies measured in this work are on the order of a few microseconds, which are orders of magnitude less than commercial Memcached latencies [125]. Low overheads are critical to preserve the advantage of hardware acceleration over software in modern datacenters [137] and enable the deployment of large memoization tables to yield better coverage. Furthermore, not all memoization opportunities are reflected at the software layer; in cases such as the video example in Section 4.2.3, blocks are reusable only when constructed on-the-fly when videos are processed. In our experiments, we saw that macroblock reuse rates across videos were insignificant, therefore having a pre-constructed software database would not be useful.

Several memoization studies were explored in the context of GPUs targeting finer-grained computations, like rendering of pixels [17], approximate scatter/gather patterns [146], caching of texture computations [167], or using ReRAM based CAMs for fine-grained computations [84]. Compared to these works, COREx supports coarse-grained memoization by combining a comprehensive design flow of large tables and emerging memories that open up new opportunities for memoization. Since COREx does not rely on internal structures (e.g., GPU shader units), it offers a solution that applies to a wide range of accelerated applications.

PIM / NVM-Based Accelerators: The ubiquity of big-data workloads and progress made in die-stacking manufacturing technologies have motivated the re-emergence of processing-in-memory (PIM) architectures, originated in the 1990s [68,134]. Modern PIM architectures reduce the costs of data movement between logic and memory dies by vertically stacking memory and logic layers on the same die (“3D stacking”) [110], or packaging multiple dies on the same silicon substrate (“2.5D stacking”) [94].

The emergence of new memory models has motivated researchers to explore new avenues for using memory to accelerate big-data applications. Recent studies explored an
in-situ acceleration of neuromorphic computations [38,149,158] combinatorial optimizations [31] and array operations [195].

While some PIM accelerators rely on emerging memory technologies, they tackle the “memory wall” problem that is induced by the high costs of memory accesses in traditional architectures. In contrast, COREx uses scalable memory technologies to replace logic, rather than bringing memory closer to logic. As we compared with PIM accelerators in Section 4.4, the gains from memoization and PIM-based acceleration are independent and can be combined.

2.4 Reused-Driven Specialization for Accelerator-Less Clouds

In Chapter 5, we present REDS, and propose it as the first programmable, reuse-driven specialization, using CPUs in clouds. To highlight REDS’ contribution in a proper scope, we discuss prior related studies and summarize their relevant properties in Table 2.2.

**Computation-Reuse/Memoization:** Early works explored fine-grained memoization of floating-point and multimedia operations [11,42]. Later works explored computation-reuse for datacenter accelerators using still-scaling memories to combat the end of Moore’s law [63], and in embedded processors to eliminate redundancies in computer-vision applications [36], DNN processing [79,143], in CPU instruction sequences [131], pure functions [165], compiled-libraries [9] and in GPU pipelines [15,17,179].

**Programmable Acceleration:** Prior works identified basic computation patterns that dominated the behavior of accelerated workloads. They also studied how the workloads affect the way accelerator architectures are built (e.g., data reuse, concurrency, etc.). Using these insights, these work aimed at improving the scope of accelerators beyond domain specialization [126,127]. In addition, prior work strived to improve the efficiency of general-purpose processors using specialized programmable cores [128].
Table 2.2: Properties and Scopes of Related Studies.

<table>
<thead>
<tr>
<th></th>
<th>Programmable</th>
<th>Reuse-Driven</th>
<th>Specialization</th>
<th>Using CPUs</th>
<th>In Clouds</th>
</tr>
</thead>
<tbody>
<tr>
<td>[11,42]</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>[105,106]</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>[126,127]</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>[128]</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>[36,79,143]</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>[9,131,165]</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>[15,17,179]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>[63]</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>[37,60,89,137]</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>REDS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Programmable Accelerators for Cloud Applications: As cloud domains strive for efficient execution, recent commercial cloud companies developed and deployed accelerators for DNNs [60,89], search engines [137], and networking fabric [37]. These works were discussed in Section 5.2.2.

Cloud-Scale Key-Value Storage: Prior works on key-value storage architectures for clouds include acceleration for Memcached [106]. In contrast to a basic key value store, REDS adds hardware for complete memoization in hardware including hardware-level input hashing, accelerated matching, and customized reuse storage.
Chapter 3

The Accelerator Wall

Specializing chips using hardware accelerators has become the prime means to alleviate the gap between the growing computational demands and the stagnating transistor budgets caused by the slowdown of CMOS scaling. Much of the benefits of chip specialization stems from optimizing a computational problem within a given chip’s transistor budget. Unfortunately, the stagnation of the number of transistors available on a chip will limit the accelerator design optimization space, leading to diminishing specialization returns, ultimately hitting an accelerator wall.

In this chapter, we tackle the question of what are the limits of future accelerators and chip specialization? We do this by characterizing how current accelerators depend on CMOS scaling, based on a physical modeling tool that we constructed using datasheets of thousands of chips. We identify key concepts used in chip specialization, and explore case studies to understand how specialization has progressed over time in different applications and chip platforms (e.g., GPUs, FPGAs, ASICs)\(^1\). Utilizing these insights, we build a model which projects forward to see what future gains can and cannot be enabled from chip specialization. A quantitative analysis of specialization returns and technological

---

\(^1\)Our modeling tool and evaluated application data are available at: https://github.com/PrincetonUniversity/accelerator-wall
boundaries is critical to help researchers understand the limits of accelerators and develop methods to surmount them.

3.1 Introduction

The ubiquity of specialized chips in modern architectures is motivated by a trisection of trends: (i) Power budget limitations imposed by the nearing end of Moore’s law and the demise of Dennard scaling. (ii) The growing computation demands of applications in battery-limited mobile devices and warehouse-scale datacenters. (iii) The emergence of compute-intensive applications that exhibit high reuse rates of computation-patterns such as matrix-vector multiplication in deep learning applications or cryptographic hashing in cryptocurrency miners. By sacrificing flexibility and targeting specific application domains, specialized chips deliver higher performance under the same power envelope. This trend has driven the increasing use of specialized IP blocks in mobile SoC platforms [150] and the deployment of accelerators in datacenters [12,47,78,89,113,137].

Chip Specialization Caveats: Knowns and Unknowns. Recent studies have explored ways to ameliorate shortcomings of specialized architectures. Frequent algorithm changes make targeted domains volatile and render specialized hardware obsolete [126], and narrowed applicability reduces the profitability of non-programmable ASIC production due to non-recurring engineering costs [96].

While the aforementioned shortcomings stem from the tradeoff between efficiency and applicability, we focus on the limitations imposed by applying chip specialization. Specialization relies on the optimization of hardware to a computation domain, subject to a given budget of power, area, and cost. Specialization has become a prominent means to prolong the evolution of chip capabilities amid the dwindling increase in transistor budget and power limitations due to the demise of CMOS scaling [56,178].
Unfortunately, the optimization space for specializing a single application is limited; there are a finite number of ways to map computational problems to fixed chip resources, and therefore, limits to the attainable chip returns for a given domain and budget. As the nearing end of CMOS scaling will cap chip transistor budgets, optimization will eventually deliver diminishing returns, and accelerators will reach a limit of near-optimal compute and hardware co-optimization. We term this limit the *accelerator wall*.

In this chapter, we conduct a study on the theoretical and empirical limits of chip specialization. Understanding these limits is the first step needed for academia and industry to create novel solutions to surmount the accelerator wall. Our paper begins with developing a methodology that quantifies the benefits of chip specialization. We then build a model for a chip’s physical potential using datasheets of thousands of manufactured chips. The potential model enables us to tease apart the end-to-end chip gains that come from advances in physical chip capabilities (e.g., CMOS technology advancements) versus the CMOS-independent benefits of optimizing a chip under a given budget. We name this CMOS-independent metric the **Chip Specialization Return (CSR) of a design**.

As a motivating example of how chip specialization progresses as applications are better understood over time, we examine the performance and CSR (how good of a job the designer did to use the transistors given) for Bitcoin mining ASIC chips as a function of time in Figure 3.1. Transistor performance is derived from the CMOS potential of evaluated chips (e.g., transistor count, CMOS speed), and the CSR is the ratio of performance/transistor performance. While performance has improved by $510 \times$, it was mainly due to a $307 \times$ improvement in transistor performance, i.e., better physical capabilities. Furthermore, the figure shows that CSR did not improve in the last two years.

We present multiple case studies to quantify the benefits of different specialization techniques and understand how specialization varies across different implementation platforms (e.g., GPUs, FPGAs, ASICs). We examine common concepts used for chip specialization, and their theoretical limits. Utilizing Aladdin [151] and our physical chip model, we con-
duct a design space exploration of specialization concepts to approximate CSR for a variety of applications. Finally, building upon the past knowledge of how CSR has progressed and how physical chip capabilities will progress until CMOS scaling ends, we build a model to project forward and explore possible future chips, quantitatively predicting the limits imposed by the accelerator wall.

Our contributions in detail include:

• We use CMOS scaling equations and datasheets from thousands of chips to construct an application-independent CMOS potential model of a chip. We use this model to decouple the contributions of CMOS technology and specialization to accelerator gains.

• We develop the metric of CSR and examine popular applications and different accelerator platforms (i.e., GPUs, FPGAs, and ASICs), to quantify how CSR has changed in accordance to the improvement in accelerator gains.

• We identify commonly used chip specialization techniques and their theoretical limitations.
• We build a modeling framework based on Aladdin [151] and CMOS equations, to tease apart the gains from specialization techniques and CMOS scaling, on a range of accelerator benchmarks [49,140,173].

• We perform a Pareto-optimal projection study for the evaluated applications, and predict the limit of accelerator potential at the end of transistor scaling.

3.2 The Interplay of CMOS Scaling and Chip Specialization Return

The Case For Specialization-Driven Roadmaps: While CMOS scaling has been the driving force of decades of exponential gains in processing capabilities, chip specialization improved the gains of specific applications or domains under a given budget. The focus on specific domains alleviates design restrictions and eliminates the structural support for unnecessary functionality [127]. We believe that future processing roadmaps and evaluation methods will become specialization-driven, focusing on specific domains [1,8] and shift away from traditional evaluation methods of benchmarking processors on a range of applications (e.g., SPEC) [50]. Similarly, new metrics will become a necessity to evaluate the quality of specialization-driven architectures.

Quantifying Chip Specialization Return: Chip specialization benefits from the co-optimization of different elements in the hardware-software stack such as the programming model and chip microarchitecture. Figure 3.2 outlines the abstraction layers of traditional architectures and our taxonomy of layers that comprise an accelerator-centric architecture. The ability to specialize a chip and improve returns, such as performance or energy efficiency, under a fixed budget is the main factor that makes accelerators relevant and appealing for future architectures. In an era of non-scaling CMOS, after transistor improvements stagnate, gains would become dependent on fixed-budget chip specialization.
We analyze chip gains. A gain is a target function the chip strives to maximize (e.g., energy efficiency) for the computation domain (the top layer). The gain is measured by executing the targeted computation on the analyzed chip, and it depends on the layers that are not fixed, i.e., Algorithm (Alg), Framework (Fwk), Platform (Plt), Engineering (Eng), and Physical (Phy). Since the complexity of chip architectures makes it challenging to measure the contribution of individual components to overall gain, we define CSR as the ratio between the gain and the gain from a chip’s physical properties (e.g., due to more transistors):

\[
CSR(Alg, Fwk, Plt, Eng) = \frac{Gain(Alg, Fwk, Plt, Eng, Phy)}{Gain(Phy)}
\]

(3.1)

The CSR metric defined in Equation 3.1 is used as a metric to quantify the merit of chip specialization, i.e.: “how much did the chip’s compute capabilities improve under a fixed physical budget?” The value of CSR improves in accordance to different components from the specialization stack, e.g.: better compilers or hardware-expressive libraries (programming framework), switching from FPGAs to more efficient ASICs (platform), ad-
vancement of chip design methodologies and RTL languages (chip engineering), and so on. By decoupling returns of chip specialization from transistor-driven benefits, the CSR can also be used to estimate the gains of future specialized chips. We use a comparative approach to estimate the impact of specialization as chip gains improve, and establish a roadmap of chip specialization trends. Given two chips, the relation between their reported gains, $Gain_A$ and $Gain_B$, their physical-driven gains, $Gain(Phy_A)$ and $Gain(Phy_B)$, and their CSRs is as follows:

$$\frac{Gain_A}{Gain_B} = \frac{CSR(Alg_A, Fwk_A, Plt_A, Eng_A)}{CSR(Alg_B, Fwk_B, Plt_B, Eng_B)} \times \frac{Gain(Phy_A)}{Gain(Phy_B)} \tag{3.2}$$

From the relations in Equation 3.2, we can examine trends of CSR for an application domain and a group of accelerators using the reported gains (e.g., energy efficiency or throughput), given a physical chip potential model (constructed in Section 3.3). These will provide us with the ability to evaluate the hardware/software co-design space for popular application domains through the following questions: (i) to what extent do specialized chips depend on the nearly-ending improvement in transistor capabilities, and to what extent do they rely on an improved specialization? (ii) when CMOS transistors stop improving, what are the projected gains for a target application before specialization reaches its limits?

### 3.3 CMOS Potential Model

We construct an application-independent model to estimate the CMOS-driven capabilities of a chip, given its physical properties. This model allows us to decouple the chip’s specialization gains from its transistor driven gains. We build the model using datasheets of 1612 CPUs and 1001 GPUs we gathered from online sources [50,171,172]. Our model receives as input the following: (i) CMOS node $(N)$, (ii) the die size $(A)$ or transistor count $(TC)$, (iii) chip operation frequency, and (iv) the chip thermal design power $(TDP)$. While
(a) CMOS Scaling. Sources: [3,4,161].

(b) Transistor Count Given Area and CMOS Node

(c) Transistor Count Given Chip Frequency and TDP

(d) Chip Gains: Throughput and Energy Efficiency for Different CMOS Nodes, TDPs, and Die Sizes for: $f_{\text{Chip}} = 1\text{GHz}$

Figure 3.3: CMOS Potential Model: Device Scaling, Chip Transistor Budget, and Physical Chip Gains.

our model is not confined to a specific target function, we focus on throughput and energy efficiency.

**Device Scaling Model:** To obtain device-level properties, we use contemporary scaling equations [161] and projections for 5nm CMOS from the recent IRDS report [4] and construct the model shown in Figure 3.3a which we use to model changes in transistor density, speed, energy, and power.

**Transistor Budget Model:** As transistor count is not always disclosed for specialized chips, we approximate the number of chip transistors. We use logarithmic regression with least mean square errors (MSE) to fit the exponential curve of transistor count as a function of chip transistor density factor ($D$), which is proportional to the die area, $A$, and inversely
proportional to the square of the node size, i.e.: \( A/N^2 \). Figure 3.3b shows the model constructed based on the datasheets. Empirically, transistor count scales sub-linearly to the density factor, since for larger chips the design complexity makes it harder to fully utilize the chip. While Figure 3.3b projects that for large 5nm CMOS chips \((D \leq 30)\), the number of transistors can reach 100 billion, not all of them can be utilized. As classic device scaling rules no longer apply to modern CMOS nodes, chips are limited by the TDP budget amid the increasing power density. Power limitations restrict the fraction of active chip transistors to keep dissipation rates within a TDP envelope \([56,178]\). Figure 3.3c shows the exponential curves of the number of transistors as a function of the chip TDP and operation frequency for different CMOS nodes. Given the TDP, CMOS node, and frequency, we use our model to derive the number of active chip transistors.

**Chip Gains Model:** We integrate the CMOS scaling model with the transistor budget models, and construct the physical chip gain model. Figure 3.3d depicts the relative scaling of the chip’s throughput and energy efficiency (executed operations per energy spent). We treat chip throughput as the targeted performance since we explore applications that possess high degrees of parallelism, like most accelerated workloads studied in literature \([23,75, 126,127]\). Gains were normalized to a 25mm\(^2\) chip fabricated with 45nm CMOS. Using the transistor budget TDP model, we show the gains for different numbers of active transistors under various power envelope zones. As reflected by the figure, power constraints cap the gains of large chips. For example, the figure shows that, while the relative throughput of an 800mm\(^2\) chip with 5nm transistors is \(\sim 1,000 \times\), under an 800W envelope, it drops by about 70\% to \(\sim 300 \times\). As expected, small chips are favorable for energy efficiency. As chips get larger, the high transistor count and static power of new CMOS nodes make old nodes more appealing under a restricted TDP.

We use the CMOS potential model to measure how specialized chips scale with respect to their transistor-driven capabilities for a given domain. The model allows us to decouple transistor-driven gains from specialization-driven gains. Finally, in Section 3.7, we com-
bine the potential model and estimated specialization-driven gains to project the attainable gains of future accelerators, ultimately reaching the accelerator wall imposed by the end of CMOS scaling.

3.4 Empirical Specialization Returns

In this section, we characterize the trends of CSR using popular accelerator domains: video decoding ASICs, graphics rendering GPUs, Convolutional neural networks on FPGAs, and Bitcoin mining using CPUs, GPUs, FPGAs, and ASICs. For each domain, we use the model from Section 3.3 to model the physical potential of the examined chips. In terms of the abstraction layers in Figure 3.2, by isolating the domain and physical layers, we examine the influence of the specialization stack layers: algorithm, programming framework, chip platform, and chip engineering.

3.4.1 ASIC Video Decoders

The footprint of digital videos has increased in recent years, due to the ubiquity of mobile devices, video sharing applications, and online services such as Netflix, whose streaming bandwidth peaks at 37% of overall internet traffic [159], and YouTube, which store hundreds of new video hours uploaded every minute [34]. Popular videos in online services can be decoded millions of times by HDTVs, smartphones, and tablets. These trends, and the spatial nature of videos, motivate the use of specialized video IP cores in commercial SoCs.

Impact of the Entire Specialization Stack: We evaluate fabricated video decoding ASIC chips and examine their gains with respect to throughput and energy efficiency. Figure 3.4 shows the scaling trends of performance, on-chip memory and core logic transistors, clock frequencies, and energy efficiency. Gains are presented in an ascending manner and are normalized to the least performing ASIC. Figure 3.4b shows estimations of the num-
Figure 3.4: Video Decoder ASICs: Performance, Hardware Budget, and Energy Efficiency. Sources: [39,41,90–92,107,174,176,203–206]

ber of transistors given the number of NAND logic gates, and the number of SRAM bits for on-chip memory and auxiliary buffers for logical units. Not all works are presented in Figure 3.4b since some works did not specify the size of on-chip SRAMs. As Figures 3.4a and 3.4c show, compared to the baseline ASIC presented in ISSCC2006 [107], absolute decoding throughput and throughput per energy improved by rates of up to 64× and 34×, 28
respectively. In contrast to throughput and throughput per energy, for the best performing ASICs, chip specialization did not improve, and even got worse since CSR was less than one. This is caused by the increase in chip transistor count (JSSC2017 has ∼36× more transistors) which enables more processing elements in parallel, and the improvement in energy efficiency for used CMOS nodes (180nm versus 40nm in JSSC2017, and 28nm in ESSCIRC2016) outpaced the improvement in overall performance and energy efficiency. Therefore, the physical layer had a higher impact than the layers in the specialization stack.

3.4.2 GPU Graphics Rendering

GPUs are the most widespread specialized chips, due to the popularity of massively-parallel workloads and maturity of programming environments such as CUDA and OpenCL. As early GPU application scope focused on graphic applications, we would like to examine the behavior of this mature domain. We analyzed a database of reported graphics benchmark results [14] and combined it with the database of GPU datasheets we used to construct the CMOS potential model in Section 3.3. Within these results, we have selected 24 popular game benchmarks including Battlefield 4, Crysis 3, GTA V, and Portal 2. Figure 3.5 shows the results of five applications (other applications show similar trends). Each of the presented applications was tested on over 20 different GPUs and normalized to the oldest GPU chip evaluated. Opaque markers show the gains of high-performance GPUs, and translucent markers show the gains of mid-end and low-end GPUs. We use quadratic curve fitting to construct curves for the reported frame-rates and CSR (frame-rate versus CMOS potential). We see that over a period of six years, performance increased by 4 – 6× (Figure 3.5a) and energy efficiency increased by 4.5 – 7.5× (Figure 3.5b). However, the CSR trends reflect that performance and energy efficiency did not improve much, if at all, beyond the CMOS potential.

Impact of Programming Framework and Chip Engineering: We evaluate the effects of advancements in GPU architectures. In terms of the specialization stack, shown
Figure 3.5: GPU Frame Rates (Apps 1-5): Throughput and Energy Efficiency

in Figure 3.2, this is a way of quantifying the benefits from: (i) Improved chip engineering, as new architectures are designed with newer tools and compilers, and using matured engineering disciplines. (ii) Better programming platforms, e.g., newer CUDA versions that support new computationally-driven libraries (e.g., video decoding, and sparse matrix computations), performance primitives, and ISA extensions [129]. We compare the gains between each pair of GPU architectures (and potentially CMOS nodes), $<$Arch,$X>$ and $<$Arch,$Y>$, using the relative geometric gains mean of all shared applications, meaning:

$$Gain(X \mapsto Y) = \sqrt[N]{\prod_{\ell=1}^{N} \frac{Gain_{<Arch,X>(App\ell)}}{Gain_{<Arch,Y>(App\ell)}}} \quad (3.3)$$

We set the relative gains of architecture pairs with at least five shared applications ($N \geq 5$), and use transitivity to compute the relative gains of architecture pairs with less than five shared applications. If relative gain between architectures $X$ and $Y$ was not set, we use the geometric means of all $M$ intermediary architectures with relations to $X$ and $Y$: 

30
Figure 3.6: Architecture + CMOS Scaling: Throughput

Figure 3.7: Architecture + CMOS Scaling: Energy Efficiency

Gain\((X \mapsto Y)\) = \(\sqrt{\prod_{\Gamma=1}^{M} \text{Gain}(X \mapsto \Gamma) \cdot \text{Gain}(\Gamma \mapsto Y)}\) \quad (3.4)

We iteratively construct the relations matrix, until we do not add a new pair. Figures 3.6 and 3.7 show the trends of performance and energy efficiency of GPU architectures and their respective CMOS nodes. We see that as expected, for a given CMOS node, newer architectures deliver better absolute gains. In terms of CSR, the first architectures to be
implemented on a new CMOS node always seem to perform worse than their predecessors on the old node (for example Fermi). As the given CMOS node stabilizes, architectures become more mature and yield better CSR. However, for both performance and energy efficiency, the overall CSR does not improve over time, as the CSR for the 16nm Pascal is roughly the same as that of the 65nm Tesla.

While newer GPU architectures tend to deliver better performance and energy efficiency, when accounting for average specialization return rates, newer GPUs do not necessarily perform better and sometimes even perform worse than their expected potential. The reason is that newer, better-performing GPUs were implemented using newer CMOS nodes with higher density and energy efficiency, therefore on average, the impact of better CMOS potential is higher than the impact of average improvement over that potential. Furthermore, maintaining even a fixed CSR is challenging since it requires new programming models and architecture capable of delivering the same relative gains for the newer and more-complex chips. Figures 3.6 and 3.7 reflect that, while GPU frame-rate gains improved by $13 - 16\times$, average CSR rates were an order-of-magnitude less, i.e., $1.0 - 1.6\times$, implying that CMOS potential is the dominating factor in the GPU graphics roadmap.

### 3.4.3 FPGA Convolutional Neural Networks

Machine learning algorithms became popular due to the increase in demands for user-personalized experience, the ubiquity of user-owned devices [141], and the increase in generated data which needs to be processed to provide a better experience on user-owned devices. The recent success of machine learning algorithms based on Convolutional Neural Networks (CNNs) in image recognition algorithms has sparked growing efforts to implement CNN image-recognition algorithms in hardware. We examine FPGA implementations of two popular models, that were notable landmarks following their performance in the ImageNet Large Scale Visual Recognition Competition [145]: (i) “AlexNet” which re-
Impact of Algorithm: Figure 3.8 shows the performance, resource utilization, and energy efficiency for FPGAs implementing both CNN models. All studies use FPGAs implemented in 28nm or 20nm CMOS. While most evaluated works explored ways to optimize the data layout and resource partitioning, some works leveraged algorithmic optimizations. FPGA17† used built-in GEMM optimizations in OpenCL and BRAM access locality to reduce the memory bandwidth requirement of CNNs. In FPGA2017* [20] the authors applied the Winograd transform [186] to exploit the locality in small $3 \times 3$ filters (used in AlexNet) and improve throughput by minimizing the complexity of convolutional opera-

---

Figure 3.8: FPGA Implementations of AlexNet and VGG-16. Sources: FPGA2016 [164], FPGA2016† [139], FPGA2016* [112], FPGA2017 [164], FPGA2017† [200], FPGA2017* [20], FPGA2018 [155], FPGA2018† [197].

Reduced the top-5 error from 26% to 15.3% in 2012 [99], and (ii) “VGG-16” which further reduced the top-5 error to 7.3% in 2014 [156].
As Figure 3.8 shows, FPGA CMOS technology had a high impact on gains, as most 20nm FPGAs were superior to the 28nm FPGAs in terms of performance and energy efficiency. While AlexNet performance and energy efficiency improved by about 24× and 14×, respectively, VGG-16 improved by about 9× and 7×, in terms of performance and energy efficiency. A source for these disparities lies in the model size. The amount of data needed to represent VGG-16 is three times the amount of data for AlexNet [153], and the amount of operations per image is about 20× [200]. The size of the model stresses FPGA resources, making it harder to optimize FPGAs for computation pattern reuse, and to run at the same clock frequencies as an FPGA implementing AlexNet. While CSR improved by up to 6× in both models, for the best performing FPGAs in each model, CSR did not improve while absolute performance increased. For these designs, performance improved due to better physical budget (higher utilization of FPGA resources). As CNNs are a relatively new domain, there is hope for new algorithms to emerge and achieve better CSR, i.e., better gains per fixed FPGA budget.

### 3.4.4 CPU/GPU/FPGA/ASIC Bitcoin Miners

Bitcoin and other cryptocurrencies made an impact on the global economy, with an equally important aspect being the energy spent to produce new currency blocks. A block stores a portion of the network transactions in exchange for fees paid to the block generator. In “proof-of-work” cryptocurrencies such as Bitcoin, blocks are added to the blockchain in a process called “mining”, which requires solving a computationally intensive problem, whose difficulty increases with the number of blocks. In the period of August-October 2018, the aggregated energy consumed by Bitcoin miners has peaked, and it was estimated at 73.12 Tera Watt-Hours annually, which is more than the energy spent by Austria [55]. While first generation miners relied on CPUs, the growing energy costs and the fact that mining computation relies on a fixed SHA-256 hash function [124] incentivized hardware specialization, and mining hardware shifted to GPUs and later FPGAs, which were quickly
overtaken by ASIC miners [169]. We constructed a mining database using datasheets and data collected from online forums to compare Bitcoin mining CPUs, GPUs, FPGAs, and ASIC chips [27–30]. As ASIC miners significantly differ in the number of integrated chips and their sizes, we treat performance per chip area as the performance metric, as it is a better indicator of chip performance than absolute throughput.

**Impact of Chip Platforms:** Figure 3.9 shows the mining gains of all tested chips. As reflected by the figure, ASIC chip gains beat CPUs by several orders-of-magnitude, as their initial CSR rate is high. Following the initial ASIC improvement, specialization return does not improve (and even declines for energy efficiency). Figure 3.9a shows how performance per area has improved by almost $600 \times$ across different ASICs (and about $600,000 \times$ compared to the baseline CPU miner), but since physical capabilities improved by $300 \times$, specialization returns improve by about $2 \times$ across ASICs. The reason for this disparity stems from the rapid transitions of one CMOS node to the next. This is demonstrated in the two distinctive energy efficiency regions, 1 and 2, in Figure 3.9b. In 1 specialization return rates improve for the early Bitcoin miners (130nm and 110nm) and in 2 energy efficiency specialization return rates improve for the modern Bitcoin miners (28nm and 16nm), but the sharp decline in specialization returns between the two regions is the result of the transition from 110nm to 28nm in a period of 15 months. This advancement outpaced Moore’s law roadmap, introducing better silicon products at a faster rate than algorithmic innovations, and therefore most benefits of ASIC miners in that period are attributed to better chip physics. This is the result of the nature of Bitcoin mining eco-system. Initially, inexpensive platforms were used, but following the increase in difficulty, miners moved to expensive ASICs with new energy efficiency CMOS nodes, since the energy spent became the dominating factor for mining revenues.

### 3.4.5 Observations and Insights

We summarize our insights from the conducted studies.
Figure 3.9: Bitcoin Mining Capabilities of CPU, GPU, FPGA and ASIC chips (vs. AMD Athlon 64 CPU Miner).

**Specialization Returns and Computation Maturity:** From our experiment, it was apparent that for mature computation domains like video encoding or gaming frame-rates, specialization returns either plateau or drop for high performing chips. This is expected, because while physical capabilities exponentially scaled in the last decade, algorithmic innovation did not scale in a similar fashion for well-studied problems. For emerging applications such as CNNs, shown in Section 3.4.3, the counter phenomena can be seen. As this domain is gaining traction, with multiple academic papers exploring a variety of algorithms on similar chips (20nm and 28nm FPGAs), we see improved CSR, by it will likely plateau as domain becomes more mature.

**Introduction of a New Specialization Platform Delivered a Non-Recurring Boost in Gains:** As seen in Figure 3.9, most CSR gains were obtained by the transition to a new platform (e.g., from FPGA to ASIC). However, following that transition, CSR did not significantly improve, and gains were mainly attained via better CMOS capabilities.

**Confined Computations:** The stagnation of specialization returns for all platforms (i.e., GPUs, ASICs, etc.) in Figure 3.9 emphasizes the challenge of algorithmically innovating a confined domain such as Bitcoin mining. Aside from ASICBoost [76] that
delivered a one-time 20% improvement by parallelizing the inner and outer loops in the algorithm, most miners operate in a brute-force and parallelized manner. Following the end of CMOS scaling, confined domains such as Bitcoin mining will become bound by the limited number of ways to represent the core algorithm in hardware.

**While Motivated by Transistor Limitations, Specialized Chips Still Highly Depend on Transistors:** In all experiments and for all chip specialization types, physical capabilities had a high impact on gains. When CMOS scaling ends, specialization improvements will slow down, and gains will remain solely dependent on improving specialization returns, that empirically scale more modestly. Therefore, the trends of CMOS and specialization returns across accelerators serve as a good indicator to assess the limits of specialization for a target domain before reaching the accelerator wall.

### 3.5 Chip Specialization: Concepts

The idea of chip specialization suggests that the knowledge of a computation domain makes it possible to couple computations from the domain layer with underlying structures implemented using the physical layer. By employing only compute-essential hardware structures, chip specialization circumvents the inefficiencies of general-purpose hardware [23,75,127]. In this section, we discuss the techniques to achieve this goal, and conduct an exploration of the chip specialization design space, focusing on its theoretical limits.

#### 3.5.1 Chip Specialization Concepts

We formalize the process of chip specialization and its relations to the three processing components: memory, communication, and computation. We observe three concepts of chip specialization, each can be applied independently to a component, or across multiple components.
**Simplification:** A narrow problem domain space provides architects with the ability to reduce the complexity of hierarchies and datapaths and attain simpler structures with similar functionality for reduced costs. Simplification can be structural (e.g., narrowing underutilized buses and datapaths for problems with limited variable representation), functional (e.g., removing floating-point units for integer problems), or control, e.g., removing energy costly out-of-order (OoO) pipe control.

**Partitioning:** Typical accelerated applications were shown to possess high degrees of parallelism [126,127]. This property motivated the development of concurrent hardware designs with replicated paths that operate independently on sub-portions of the application data. Common forms of partitioning are SIMD/VLIW vectorization, threaded parallelism, and the bisection of on-chip mesh networks-on-chip (NoC).

**Heterogeneity:** Even a single workload has diverse needs: certain computations require non-trivial functionality. A workload might possess phases with different computation intensities or reuse numerous distinct patterns. While simplified and partitioned hardware improves efficiency for these cases, further reduction of energy per computation is possible by diversifying the computation paths and tailoring each path to support specific functionality. Common forms of heterogeneity are fusing of functional units (e.g., clustering of instructions [69] or dataflow graph phases [72]), algorithm-specific functional units (e.g., tangent activation units for neural networks), and asymmetric memory banks and network topologies to accommodate irregular data patterns.

Nowatzki et al. [126] suggested a taxonomy of principles: concurrency, communication, data-reuse, computation, and coordination, that can be mapped to the concepts we define. We found our classification a better fit for this limit study.

**Case Study: Tensor Processing Unit:** The increase of user-generated sensory data necessitates high processing power. Google engineers designed a 28nm ASIC chip called a Tensor Processing Unit (TPU) to avoid from having to double their CPU-based datacenters to meet the demands of speech recognition alone [89]. They demonstrated how TPUs im-
Table 3.1: Chip Specialization Concepts. Examples From a TPU ASIC Chip.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Simplification</th>
<th>Partitioning</th>
<th>Heterogeneity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple DDR3 chips,</td>
<td>Simple DDR3 chips, interfaces, and physical memory space</td>
<td>Memory module banking storing NN layer weights</td>
<td>Hybrid memory for input and intermediary results</td>
</tr>
<tr>
<td>and physical memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>space</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication</td>
<td>Simple FIFO communication</td>
<td>Concurrent FIFOs for weights and systolic array data</td>
<td>Software-defined DMA Interface for chip I/O</td>
</tr>
<tr>
<td>Computation</td>
<td>Multiply+add computation units with small precision (8-bit integers)</td>
<td>Parallel multiply+add paths</td>
<td>Non-linear activation unit (e.g., ReLU) and systolic array data reuse</td>
</tr>
</tbody>
</table>

prove the energy efficiency of deep neural network (DNN) workloads by $80 \times$ compared to CPUs. Figure 3.10 shows a simplified block diagram of a TPU, with annotated examples of all specialization concepts, summarized in Table 3.1.

### 3.5.2 The Limits of Chip Specialization Concepts

We discuss the limits of chip specialization concepts. We present the target computation problem as a dataflow graph (DFG). A DFG is a concise representation of computation problems, limited solely by inherent computation restrictions (e.g., data dependencies), and not by implementation mediums (e.g., timing, area, or power restrictions). This property
makes DFG optimization a useful way to model the design space visible to the specialization stack layers in Figure 3.2.

The DFG is a directed-acyclic graph (DAG), $G(V, E)$, in which $V$ is the set of vertices: $V = \{v_1, v_2, \ldots, v_{|V|}\}$ and $E$ is the set of edges: $E = \{e_{v_i, v_j} | v_i, v_j \in V\}$. The DFG succinctly describes the high-level relations of data and computation, and how data flow across vertices.

Definitions:

- **Input variables set** is a set of vertices with no incoming edges: $V_{IN} = \{v_{in_1}, \ldots, v_{in_N}\} \subseteq V$.

- **Output variables set** is a set of vertices with no outgoing edges: $V_{OUT} = \{v_{out_1}, \ldots, v_{out_M}\} \subseteq V$.

- **Computation nodes set** is a set of vertices with incoming edges and outgoing edges, that represent computation operands: $V_{CMP} = \{v_{cmp_1}, \ldots, v_{cmp_M} | v_{cmp_i} \notin V_{IN}, v_{cmp_i} \notin V_{OUT}\} \subseteq V$.

- **Computation paths set** is a set of all graph routes, i.e., vectors of edge-connected vertices, starting with an input variable and ending with an output variable: $P = \{(v_{p_1}, \ldots, v_{p_K}) | e_{v_{p_i}, v_{p_{i+1}}} \in E, v_{p_1} \in V_{IN}, v_{p_K} \in V_{OUT}\}$.

- **The DFG depth** is length of the longest computation path, i.e.: $D = \max\{K | (v_{p_1}, \ldots, v_{p_K}) \in P\}$.

- **Computation stage working set** is the set of variables computed in computation stage $s$, i.e.: $WS_s = \{v_{p_{1,s}}, \ldots, v_{p_{n,s}} | (v_{p_{1,1}}, \ldots, v_{p_{1,s}}, \ldots) , (v_{p_{n,1}}, \ldots, v_{p_{n,s}}, \ldots) \in P\}$.

Figure 3.11 shows a DFG with three input variables, two computation stages, and two output variables. We use DFG definitions to explore hardware optimization bounds, and consequently, chip specialization limits. For completeness, we do not account for hardware limits (e.g., unlimited area).
**Memory Simplification:** Memory optimizations target storage and memory access costs. Memory simplification reduces storage at the expense of access performance. The simplest hierarchy consists of a single module that stores only needed variables. The minimal state that needs to be stored in the memory hierarchy is the number of variables processed at a given time. Therefore, it is bound by the largest working set size: $\Theta(\text{max}|WS_i|)$. For simplicity, and without loss of generality, we assume fixed variable sizes, hence reading or writing has $\Theta(1)$ cost, and access costs depend on the cost of lookups. Lookup is bounded by the variable naming space. Therefore, lookup costs are at least $\Theta(log(\text{max}|WS_i|))$ for a single variable. For each stage, the stage’s computation nodes are accessing the memory sequentially, since the simplest memory interface does not support parallelism. Since all nodes have to access the memory, timing complexity is, therefore, $\Theta(|V| \cdot log(\text{max}|WS_i|))$.

**Memory Heterogeneity:** We express memory heterogeneity as a hierarchy formed by a layout of modules and/or interfaces to support problem-specific access patterns. Maximal performance is achieved by a banked hierarchy that reflects all computation relations across nodes. Since relations are described as DFG edges, storage costs are on the order of $\Theta(|E|)$.
The hierarchy performs fast $\Theta(1)$ accesses, done in parallel in each stage. Total timing complexity is on the order of stages, or DFG depth, i.e., $\Theta(D)$.

**Communication Simplification:** A communication fabric is simplified by reducing the number of wires at the expense of increased latency. The limit is a minimal spanning tree connecting all nodes. Further reduction of wires would leave unconnected nodes, therefore the minimal wire complexity is $\Theta(|V|)$. Since data must traverse across all nodes to satisfy all dependencies defined by the edges, the timing complexity, in terms of network hops, is $\Theta(|E|)$.

**Communication Heterogeneity:** Since the DFG topology is structured in a way that reflects the problem-specific communication patterns, communication heterogeneity is explicitly defined by the DFG edges. The wiring complexity is on the order of network edges, i.e., $\Theta(|E|)$. The timing complexity is the DFG delay, i.e., the number of stages in the longest computation path, or DFG graph depth $\Theta(D)$.

**Computation Simplification:** DFG nodes express problems using pre-defined operands (e.g., ADD, MUL). As computation completeness dictates that mathematical operations can be computed using logic gates, a node is reduced to a set of $\Theta(1)$ gates. For fixed-size variables, operands are reduced to a fixed number of gates, and computation is done in a serial bitwise manner for each input-output pair. Therefore, the total timing is the number of nodes $\times$ number of input variables $\times$ number of variable bits, which is on the order of all edges that express the relations between the problem nodes, or $\Theta(|E|)$.

**Computation Heterogeneity:** Computation heterogeneity is done by fusing nodes to form problem-specific “super nodes”. The extreme case for fusing is a single node which acts as a lookup table that stores the computation results of all computation inputs. The table would have $\Theta(2^{|V_{IN}|})$ entries (total input bits). Each entry contains the computation result, which is on the order of the number of output variable bits, i.e., $\Theta(|V_{OUT}|)$. The total space complexity is, therefore: $\Theta(2^{|V_{IN}|} \cdot |V_{OUT}|)$. The time complexity consists of the time for lookup $\Theta(log(2^{|V_{IN}|})) = |V_{IN}|$ and reading the computation output, i.e., $\Theta(|V_{OUT}|)$, and
Table 3.2: Summary of Time and Space Complexity Limits for Chip Specialization Concepts, in Terms of DFG Definitions.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Time</th>
<th>Heterogeneity</th>
<th>Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simplification</td>
<td>Heterogeneity</td>
<td>Partitioning</td>
</tr>
<tr>
<td></td>
<td>$\Theta(</td>
<td>V</td>
<td>\cdot \log(\max</td>
</tr>
<tr>
<td></td>
<td>$\Theta(</td>
<td>E</td>
<td>)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Communication</th>
<th>Time</th>
<th>Heterogeneity</th>
<th>Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Theta(</td>
<td>E</td>
<td>)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Computation</th>
<th>Time</th>
<th>Heterogeneity</th>
<th>Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Theta(1)$</td>
<td>$\Theta(2^{</td>
<td>V_{IN}</td>
</tr>
</tbody>
</table>

| the total is: $\Theta(|V_{IN}| + |V_{OUT}|)$. Partitioning the table to $|V_{OUT}|$ tables, each storing an output variable, would reduce output variable read time to $\Theta(|1|)$, resulting in a time complexity of $\Theta(|V_{IN}|)$, the same time as reading all inputs ($\Theta(|V_{IN}|)$), solving computation in $\Theta(1)$ runtime, and writing all outputs in parallel in $\Theta(1)$ runtime. | Memory / Communication / Computation Partitioning: Partitioning is limited by the level of DFG parallelism, which is on the order of the maximum number of concurrently processed variables. This is the largest working set size, i.e., $\Theta[\max|WS_s|]$, since further partitioning would produce diminishing returns. The total space complexity for a maximally partitioned system is, therefore $\Theta[\max|WS_s|]$, and since the maximal path stages is: $\Theta(D)$, the lookup time for variables in the memory takes $\Theta(D \cdot \log(\max|WS_s|))$, and communication and computation time complexity is $\Theta(D)$. |

Table 3.2 summarizes the limits of the discussed chip specialization concepts. As chip specialization is conceptually limited, optimization space is finite and is further restricted when combined with realistic hardware limitations.

### 3.6 Gains of Chip Specialization Concepts

While Section 3.4 explores the empirical gains of real-world accelerators, the sources of specialization concepts that contribute to the obtained gains are not disclosed and should be quantitatively explored. We quantify the contributions of chip specialization and CMOS potential for a range of applications. We determine the optimal points for each applica-
Table 3.3: CMOS-Specialization Sweep Parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explored Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning Factor</td>
<td>1, 2, 4, ... 524288</td>
</tr>
<tr>
<td>Simplification Degree</td>
<td>1, 2, 3, .., 13</td>
</tr>
<tr>
<td>CMOS Process (nm)</td>
<td>45, 32, 22, 14, 10, 7, 5</td>
</tr>
</tbody>
</table>

...tion, attribute the sources of gains from CMOS savings (i.e., more energy-efficient CMOS nodes) and the specialization concepts described in Section 3.5.

**Methodology:** Our framework is integrated with Aladdin, a modeling tool that enables fast exploration of accelerator design alternatives [151]. The original Aladdin flow supports partitioning of memory and datapaths using unrolling, pipelining and memory banking. It also models heterogeneity using DMA and scratchpads, register layout, and fusion of operands for dependent dataflow graph nodes that fit the same cycle. We extend the Aladdin flow to support CMOS scaling using our model described in Section 3.3. We add support for simplification and pipelining of functional units and registers, using data from [65]. By applying these techniques, we approximate the hardware/software co-optimization of accelerator applications using the specialization concepts described in Section 3.5. These concepts are also implemented in modern accelerators, e.g., Hyper-Pipelining in Stratix 10 [85], and computational heterogeneity in the Xilinx ACAP model [192]. Table 3.3 lists the explored parameters, and Table 3.4 summarizes the evaluated applications.

**Case Study: Stencil Computation:** 3D Stencil can be used to extract spatial image features for computer vision applications [138]. As Figure 3.12 demonstrates, the Stencil kernel is highly parallel, as filtering can be applied concurrently to different members of the “Orig” lattice. We explore different design alternatives for a 3D Stencil accelerator, shown in the Runtime-Power space in Figure 3.13. As expected, CMOS advancement reduces power. As the figure shows, partitioning improves performance gains for all technologies, until reaching a point where runtimes plateau, when reaching the maximal degree of kernel...
parallelism. Following that point, old nodes (e.g., 45nm) experience diminishing returns due to underutilized partitioned resources. However, performance still improves for newer CMOS nodes, since functional units are faster, and more computation units are fused and scheduled in a cycle. This is a combination of computation heterogeneity with advanced CMOS processes. Simplification of functional units, registers, and communication, also reduces power (as smaller structures generate less leakage power). The optimal points for energy efficiency are received for 5nm CMOS, for the highest degree of partitioning for which performance does not taper off, and the highest simplification degree that does not cause diminishing returns (i.e., increased latency due to deep pipelining).

We sweep the design space for the evaluated applications to determine the contribution of CMOS and specialization concepts. For each application, gains were normalized to a 45nm accelerator with no simplification or partitioning. Figure 3.14 shows the obtained gains. As simplification and CMOS power saving reduce energy and not runtime, they improved efficiency but not performance. While partitioning was the primary contributing source for performance, CMOS saving was the dominating factor for energy efficiency. For
both performance and energy efficiency, CSR is low, since both CMOS saving and partitioning (i.e., using more transistors for parallelization) are inherently CMOS dependent.

3.7 The Accelerator Wall

We explore the limitations of the applications evaluated in Section 3.4. The motivation for this limit study stems from the dominance of CMOS-driven capabilities on accelerator
gains. As accelerators are CMOS dependent, we use the CMOS-driven potentials to estimate the attainable gains of each domain and project the accelerator wall, which is: the best performance and energy efficiency for accelerator chips manufactured after CMOS technology stops scaling.

Physical Chip Limits Parameters: We use the CMOS scaling equations and chip model in Section 3.3 to estimate the accelerator gains for chips implemented in the final CMOS node, currently projected to be 5nm [4]. Table 3.5 summarizes the explored parameters in each evaluated domain. We follow the insights from Section 3.3, and use largest dies for performance, and smallest dies for energy efficiency. For FPGAs, we used typical TDP and die size numbers reported for Xilinx FPGA boards [191]. For video encoding
ASICs we use a TDP budget of 7W, 10× higher than the highest power measure of 690mW in [203].

**Projection Models:** We examine two projection models for the evaluated accelerators. (i) The *Linear* model is a Pareto frontier projection that assumes a linear connection between physical capabilities and the chip gains:

\[
\text{Projection}_{\text{Linear}}(\text{Physical}) = \alpha \cdot \text{Physical} + \beta
\]  

(ii) The *Logarithmic* model is a Pareto frontier projection that assumes a sub-linear connection between physical capabilities and gains.

\[
\text{Projection}_{\text{Log}}(\text{Physical}) = \alpha \cdot \log(\text{Physical}) + \beta
\]  

Our models account for the difference in application behavior, domain and silicon maturity, and specialization platforms. For example, GPUs rely on massive parallelism, and are likely to exhibit linear behavior with regards to physical performance (more transistors map to more cores). The sub-linear logarithmic projections reflect the difficulty in exploiting high complexity chips (e.g., due to peripheral overheads), or due to inherent (e.g., structural) algorithmic limitations in the ways the domain can be specialized using the added silicon budget.

**Results:** Figures 3.15+3.16 show the projections of performance and energy efficiency for the evaluated accelerators. Generally, the linear model fits the performance spaces, and the logarithmic model fits the energy efficiency spaces. Since accelerated applications possess high parallelism [127], performance scales linearly by adding more parallel processing elements. In contrast, energy efficiency requires simplifying datapaths and exploring other hardware tradeoffs. For example, while Figure 3.15b shows the performance Pareto fron-
Figure 3.15: Evaluated Applications and Specialization Chips: Accelerator Performance Projections.

tiers for different GPUs and games behave linearly, that is not the case for Figure 3.16b. The high disparity of accelerator gains obtained under similar physical capabilities for GPUs is caused by the variety of different applications evaluated and targeted devices (e.g., server vs. smartphone GPUs).

**Accelerator Limits:** According to our limit study and given the available chip data, while ASIC video decoding performance and energy efficiency improved by $64\times$ and $34\times$, respectively, we project further performance and energy efficiency improvements of $3 - 130\times$ and $1.2 - 14\times$. While GPU graphics frame rate improved by a rate of $16\times$, we project further performance and energy efficiency improvements of $1.4 - 2.5\times$ and $1.4 - 1.7\times$, respectively. While FPGA CNN performance improved by $22.5\times$, we project further performance improvements of $2.1 - 3.4\times$ and energy efficiency rates of $2.7 - 3.5\times$. 

49
While ASIC Bitcoin performance increased by $600 \times$ (and about $600,000 \times$ over CPUs), we project further improvements of $2 - 20 \times$ and $1.4 - 5 \times$ in performance and energy efficiency, respectively. All the limits were obtained using projections of 5nm CMOS technology, which is not commercially available yet.

Our study shows that while performance has a promising trajectory for most domains, energy efficiency is not projected to improve at the same rate. Although specialization is hard to predict, it is generally easier to predict domains that are mature, well-studied, and algorithmically-stable. For example, GPU frame rate is easier to predict than FPGA CNNs, since the analyzed CNN data span only four years of specialization efforts. Interestingly, the properties that make specialization predictable are also the same ones that make domains fit for specialization in the first place [96,126,127,178]: mature domains
have a variety of adopted applications, and well-studied domains are stable and less prone
to volatility and specialized hardware obsoletion. As domains mature and dominating al-
gorithms as stabilized, so will chip specialization, and it will become harder to improve
gains under a fixed chip budget. Following the end of CMOS scaling, accelerator gains
will become bound by the reach of specialization, which can be quantified and projected
using the metrics and methods presented in this work.

3.8 Conclusions

While the end of transistor scaling has motivated a shift to specialized architectures, spe-
cialized chips still greatly depend on gaining more transistors. In this work, we conducted
a careful limit study of chip specialization. In order to isolate and remove the benefits
of device technology, we built a CMOS potential model using datasheets of thousands of
real-world chips. We characterized how specialization returns have progressed in popular
chip-specialized application domains. We demonstrated that while domain maturity and
stability motivate hardware specialization, they also result in diminishing specialization re-
turns. We identified common chip specialization concepts, presented their theoretical limits
and evaluated their contribution to application gains. Finally, we conducted a limit study
that projected the accelerator wall of future potential chips for each application. The limits
imposed by the accelerator wall and the evaluation methodologies presented can be used
as foundations for a better understanding of future accelerated applications. These will
become a necessity for future designs in the era of accelerators and non-improving CMOS
technology.
Chapter 4

COREx: A Compute-Reuse Architecture for Datacenter Accelerators

Hardware specialization is commonly used in datacenters to ameliorate the nearing end of CMOS technology scaling. While offering superior performance and energy efficiency returns compared to general-purpose processors, specialized accelerators are bound to the same device technology constraints, and are thus prone to similar limitations in the future. Once technology scaling plateaus, accelerator and application tuning will reach a point of near-optimum, with no clear direction for further improvements.

Emerging non-volatile memory (NVM) technologies follow different scaling trends due to different physical properties and manufacturing techniques. NVMs have inspired recent efforts of innovation in computer systems, as they possess appealing qualities such as high capacity and low energy.

We present the COmpute-REuse Accelerators (COREx) architecture that shifts computations from the scalability-hindered transistor-based logic towards the continuing-to-scale storage domain. COREx leverages datacenter redundancy by integrating a storage layer
together with the accelerator processing layer. The added layer stores the outcomes of previous accelerated computations. The previously computed results are reused in the case of recurring computations, thus eliminating the need to re-compute them.

We designed COREx as a combination of an accelerator and specialized storage layer using emerging memory technologies, and evaluated it on a set of datacenter workloads. Our results show that, when integrated with a well-tuned accelerator, COREx achieves an average speedup of $6.4 \times$ and average savings of 50% in energy and 68% in energy-delay product. We expect further increase in gains in the future, as memory technologies continue to improve steadily.

4.1 Introduction

Specialized hardware such as GPUs and domain-specific accelerators have become increasingly popular over the past decade. Hardware specialization represents the shift towards alternative computing models, which is motivated by the end of Moore’s Law for CMOS technology scaling, currently projected by the ITRS to take place in 2021 [3]. Hardware specialization couples domain-specific accelerators and applications to deliver higher performance within a given power envelope or for the same energy rates. This positions accelerators as an appealing building block for future datacenters and has sparked the emergence of datacenter accelerators in both industry [47,89,137] and academia [93,106,113] which was made possible by changes to the software stack [5,47].

Unfortunately, the implications of the end of Moore’s Law are not solely restricted to general-purpose processors, as accelerators are also implemented using transistors. Since hardware specialization is limited by the number of ways that exist to map computation problems to a fixed hardware budget, accelerator gains cannot improve beyond a certain point. When transistors have stopped scaling, and all available transistors are specifically tailored to a given computation, additional optimizations will not be possible. The pressing
question at this point should be: *can systems keep improving their computing capabilities, even when the number of transistors has stopped increasing?*

In contrast to the nearing end of CMOS scaling, recent advancements in emerging memory technologies have shown promise for future scaling. Compared to traditional DRAMs, Phase-Change Memories (PCMs) have superior storage due to the ability of storing multiple bits in a single memory cell [102,188,196,202], Resistive-RAM (ReRAM) crosspoint structures have superior density [10,193] since cells scale to small feature sizes [108,115], Spin-Transfer Torque RAM (STT-RAM) has been shown to have superior performance while reducing energy costs [86,100], and the density and energy efficiency of Magnetic Domain-Wall Racetrack memories motivated its use in on-chip memories [177].

An important observation for datacenter workloads is that in many scenarios, datacenters process the same computed data. Internet traffic has been empirically shown to follow skewed Zipfian distributions [7,33]; video encoders in services like YouTube compress identical blocks stemming from spatial redundancy in the encoded video frames; search engines fetch the same popular pages for different search terms, and access patterns were also shown to have skewed distributions in cloud services such as media [123] and social networks [121].

The re-manifestation of the same computed data introduces redundancies in accelerator execution, which could (and should) be avoided, by leveraging mechanisms for memoization of redundant computations [118]. Given the correct storage to compute cost ratio, it is possible to record an accelerated kernel’s inputs and results and store them. In the case that a previously encountered input is fed into an accelerator, the pre-computed result is fetched from a storage device instead of being re-computed by the accelerator, therefore saving transistor-based compute and energy.

We present COREx, an architecture that employs emerging memory technologies as an auxiliary storage layer for the purpose of memoization. The COREx architecture builds a customized memory hierarchy, optimized for the accelerator. By shifting recurring com-
Computations from accelerators to the memory domain, **COREx essentially trades technologically non-scaling transistor logic for still-scaling storage.** Figure 4.1 shows how compute-reuse storage is embedded as an extension layer to the acceleration fabric, consisting of loosely-coupled accelerators (LCAs) [44,152]. LCAs and host processors have isolated memory spaces and explicitly communicate via DMA transactions, making LCA designs a natural target for memoization.

We explore the use of ReRAMs, PCMs, Racetrack memories, and STT-RAMs for memoization tables. We show how COREx can be optimized for different goals: runtime, energy, and energy-delay product (EDP). The contributions of this chapter are as follows:

- We present a discussion on workload redundancy in datacenter accelerators, quantitatively focusing on two case studies: video encoding and traffic compression in search engines.

- We develop a framework for accelerator memoization, discuss tradeoffs, and present an end-to-end flow that specializes compute-reuse memory layers for a variety of accelerated computations and optimization goals.
• We design the COREx prototype, and tune it separately for performance, energy, and EDP. We show that compared to well-tuned accelerators, COREx achieves an average speedup of $6.4 \times$, consumes 50% less energy, and reduces EDP by an average of 68%.

### 4.2 Background and Motivation

This section outlines the trends that motivated the development of COREx, and guidelines for applying scalable memoization in the datacenter.

#### 4.2.1 Specialization and Scalability

The inevitable end of Moore’s Law and Dennard scaling renders traditional processor performance and energy scaling models obsolete and motivates a shift away from existing computing paradigms to achieve sustainable technological growth [56]. Specialized hardware became the focus of recent studies that integrate new programming models with domain-specific accelerators to deliver more efficient compute capabilities compared to general-purpose processors under a given hardware budget.

Unfortunately, the end of CMOS scaling that motivated specialization also implies that specialization gains cannot continue growing. Specialization relies on co-optimization of computation problems and hardware to achieve an efficient mapping of the computation problem to a given transistor budget. The gains from mapping algorithms to given hardware are capped since the optimization space is not unlimited, and when hardware stops scaling, co-optimization gains cannot exceed the point of near-optimum. This will necessitate a shift away from direct dependence on the non-scaling transistor technology in future accelerators.

In contrast to CMOS process technology, modern memory and storage systems continue to scale. Figure 4.2 depicts traditional scaling trends, and uses contemporary projections to predict future technology scaling curves. For transistor density scaling, we
used existing CPU datasheets from past databases [50,171] and extrapolated the curve following the current predictions of the International Technology Roadmap for Semiconductors (ITRS) [3] for the end of Moore’s law in 2021. For accelerator adoption in high-performance computers (HPCs), we use the statistics of six recent years from the Top500 list [175], when it started listing accelerators and co-processors. For storage density scaling, we used the ITRS “More Moore” technology roadmap for NVMs, and data from existing academic [109] and commercial devices [168].

4.2.2 Memoization Requirements: Three C’s

Proper memoization designs must meet several criteria. We term them as the Three C’s for Memoization:

1. **Correct Results**: Memoization must preserve computation semantics by always producing the same result as the non-memoized computation, a property known as “referential transparency” [162]. This property requires the construction of a generalized input
set that consists of every input and element in the system state that can affect the computation outcome. For example, the search results for the term: “where am I” fed as input to a search engine depend not only on the input itself, but also on additional state such as the inquirer’s location. This suggests that search engine memoization solely based on the search term might not produce correct results, as the correct result is different for distinct locations. As we show in this chapter, memoization can be naturally done for loosely-coupled accelerators. Since the relation between computation results and input is deterministic and well-defined, correctness is guaranteed by design. This makes memoization of accelerated kernels natural and appealing compared to datacenter memoization which is software-based and requires ad-hoc program adaptations or employing particular software frameworks [35,73,144].

**Constrained Storage:** While storage scaling rates show promise for the coming years, we strive to keep storage tables constrained to ensure a feasible implementation of the storage layer. In order to achieve confined memoization storage, one must understand how computation input values vary. If the input variance is high, the amount of information that needs to be stored might exceed the available storage.

**Costworthy Reuse:** The last requirement is the profitability of memoization. The cost of the memoized computation should be greater than the costs of storing, looking up, and fetching computation results. This requires an understanding of tradeoffs in the compute and compute-reuse design spaces. For instance, large reuse tables yield better coverage, however, as table size increases, access times and energy increase accordingly. Depending on memoization granularity, input behavior, and the mapping of computation to reuse tables, memoization might, or might not, be of gain.

We formulate the prerequisites for memoization profitability using a general cost model. The cost function (e.g., runtime) for each operation \(x\) is marked as: \(C_x\).
\[ C_{\text{compute}} \geq C_{\text{lookup}} + P_{\text{hit}} \cdot C_{\text{hit}} + (1 - P_{\text{hit}}) \cdot C_{\text{miss}} \]

\[ C_{\text{miss}} = C_{\text{compute}} + C_{\text{table\_update}} ; \quad C_{\text{hit}} = C_{\text{table\_read}} \]

Memoization is therefore profitable if the following holds:

\[
\begin{align*}
P_{\text{hit}} & \geq \frac{C_{\text{lookup}} + C_{\text{table\_update}}}{C_{\text{compute}} + C_{\text{table\_update}} - C_{\text{table\_read}}} ; \quad 1 \geq P_{\text{hit}} \geq 0
\end{align*}
\] (4.1)

### 4.2.3 Reuse Opportunities in Datacenters

The increase in datacenter energy consumption has sparked changes in every layer of the datacenter stack in search of efficient means of processing. This trend leads to growing efforts of designing and integrating accelerators in commercial datacenters [47,89,137].

While accelerators enable energy-efficient datacenter processing capabilities by specializing for common computation templates (e.g., machine-learning computations and graph-processing), recurring data patterns in datacenter workloads provide an additional opportunity for further efficiency.

**Case Study I: Web-Server Traffic Compression:** The steady increase in client-server traffic has brought traffic compression to the frontline of web-server designs. Recent surveys report that roughly 70% of all websites use gzip or one of its variants for traffic compression [166]. Gzip is supported by Apache and has been used as the baseline for Snappy [71], which is Google’s compression library for its datacenters, big data processing machines, and internal remote procedure calls (RPCs).

Recently, there have been several proposals for embedding datacenters with hardware implementations of gzip in ASIC and FPGA [52,130]. We present a case where memo-
Figure 4.3: Traffic Reuse and Storage Requirements across 100K Web Searches; (A) Retrieved versus Reused Records, (B) Retrieved versus Uniquely Stored Traffic Size.

Internationalization support for compression accelerators in datacenters is useful. In this scenario, we demonstrate the behavior of a web-based search engine that compresses result traffic. We collected a trace of database server traffic using the TailBench client-server benchmark suite [95]. The trace consists of the traffic generated by 100k search queries to a Xapian server [190] simulating a Wikipedia search engine. The generated traffic consists of records containing Wikipedia abstracts; each search query returns up to 25 records. In this scenario, it is likely that different terms will hit on the same records as part of their query result (e.g., “sign” and “language” can both retrieve the Wikipedia abstract for “sign-language”). As the number of tracked search terms grows, the fraction of previously retrieved abstracts grows accordingly.

Figure 4.3 depicts the timely mapping of: (A) the records retrieved by search queries and the number of recurring records, i.e., records fetched by more than one query, and the fraction of recurring records from overall retrieved records; (B) server traffic generated by retrieved records and the increase in capacity for a theoretical storage device that uniquely maps the traffic by records, i.e., without storing any record more than once.
The figure highlights two interesting trends: (i) since the searched database is finite, the capacity required to construct the irredundant storage device saturates when the more popular records are already mapped, therefore mapping retrieved traffic to a confined storage device is possible. (ii) Record reuse rate increases to \( \sim 80\% \), and is expected to increase further as the number of searches grows. From this experiment, we conclude that traffic compression accelerators for web search servers can gain from reuse storage.

**Case Study II: Video Encoding:** We present a different compute-reuse scenario that is not caused by recurring server traffic, but by redundant computations in video encoding. Video services have become increasingly ubiquitous over the past decade, and in specific, video uploading and sharing among different users. In 2015, the reported upload rate for YouTube servers was estimated at 400 hours of video every minute [34]. These rates establish the popularity of video encoders, making them a target for hardware acceleration [113].

Modern video encoding techniques rely on the spatio-temporal locality in distinct parts of the encoded frames; each frame is split into groups of pixels, referred to as: “macroblocks”. The locality assumption for video macroblock handling is that many videos consist of a few continuously moving subjects and a stationary background. Therefore, macroblocks that comprise the background tend to have fewer changes, while the macroblocks corresponding to moving objects can be tracked across frames in a process called: “motion estimation”. Under the assumption of continuously moving objects, video en-
coders predict the future location of the moving objects’ macroblocks in a process called: “motion compensation prediction”.

Figure 4.4 shows an execution time breakdown of the main tasks in the x264 video encoder [25], for a sequence of the ten longest videos from the YouTube Faces video dataset [187]. The breakdown statistics were gathered from profiles of multiple x264 executions on a 40-core x86 machine, using 32 threads. From the depicted breakdown, it is apparent that macroblock handling accounts for a large \( \sim 88\% \) fraction of overall runtime, and specifically, more than 60\% is attributed to motion estimation and motion prediction alone.

Interestingly, the stationary nature of macroblocks that video encoders leverage can be exploited in the context of memoization as well, due to the computational redundancy made by processing recurring stationary macroblocks. Figure 4.5 demonstrates a visualization of macroblock reuse in a video, for different macroblock sizes. As expected for small macroblock sizes (8 \( \times \) 8), after four seconds of video, 77\% of a frame’s blocks were already recorded in previous frames. As macroblocks size increases, it is harder to match recurrent macroblocks. This figure demonstrates a fundamental tradeoff for memoization, which can be applied to Equation (4.1): while potential memoization gains are higher for expensive computations with larger inputs \( (C_{\text{compute}} \uparrow) \), larger inputs are often harder to match, since they are less likely to exhibit high recurrence rates \( (P_{\text{hit}} \downarrow) \).

### 4.3 The COREEx Architecture

COREx adopts a storage-centric approach to reduce the use of transistor-based accelerators by using memoization to fast-forward redundant computations. It relies on the integration of an auxiliary storage layer to back the computation layer. In this section, we present the fundamental elements of the COREx architecture. We describe the integration of COREx
in the accelerator flow, present the main hardware components in the COREx scheme, and show the different stages of the hardware/software co-design flow.

### 4.3.1 Execution Flow

**Accelerator Coupling:** Accelerator models can be classified as Tightly-Coupled Accelerators (TCAs) and Loosely-Coupled Accelerators (LCAs) [48]. While TCAs are integrated with the CPU’s pipeline and share its caches, LCAs are equipped with private scratchpad memories and communicate with the CPU via DMA transfers. LCA designs are considered more flexible, as they are not confined to a specific CPU architecture, they do not compete with CPUs over caches, and their use of heavily-banked scratchpad memories makes them a better fit for data-intensive workloads.
**Accelerator Flow Modifications:** One of the main appeals of the LCA model is that by decoupling DMA transfers from accelerator executions, they can be orchestrated independently and benefit from DMA optimizations such as double-buffering. The decoupling of input, execution, and output makes LCAs ideal for memoization, since accelerators are relatively stateless and have no side effects on the system memory. Referential transparency is, therefore, guaranteed by design without additional workload assumptions or support from the software stack. To support memoization, we modified the original execution model. Figure 4.6 shows a sequence of three kernel computations under different execution flows: (A) the traditional model, which corresponds to a serial input → compute → output execution, (B) a double-buffering model by Cota et al. [48], which improves utilization by overlapping execution and DMA transfers, and (C) the “HLR” model: the new COREx flow of: “Hash → Lookup → Reuse”, which is integrated with the accelerator compute stage. In the example given, the first two executed kernel inputs were already recorded by previous executions, while the third kernel input is a new input. Numbers signify the stages of different HLR executions. 1 The first input is incrementally hashed as DMA transfers are fed to the accelerator (therefore the “H” and “I” stages overlap), and then the hash result is looked up in the table at the “L” stage. 2 When the input is fully hashed, the hash lookup “hits”, i.e., the hashed input matches the input fed to the accelerator, therefore the output can be fetched from the table instead of recomputing the input. 3 Once the output is fetched, in the “R” stage the computation is reused, i.e., sent back to the CPU. The input of the third kernel was not previously recorded, therefore, in 4 it results in a “miss” in the “L” phase, and the accelerator subsequently executes. We also examined a policy in which accelerator execution speculatively overlaps with the lookup and aborts on lookup hit, but we found that the runtime gains of this approach were insignificant compared to the added energy costs.
Figure 4.6: Accelerator Execution Flavors.

4.3.2 Top-Level Design

To support the HLR execution model, we integrate new hardware structures with the existing LCA hierarchy, as shown in Figure 4.7. Since storage tables might be large, table accesses may have high timing and energy costs. Therefore, we designed the COREx architecture to have a two-phase lookup. A lookup of the hashed input in a small cache, and a full input comparison using entries stored in the large, RAM-like table. Without the first lookup phase, the flow still works correctly but might have more unnecessary accesses to the large table. We added three structures to the baseline accelerator; (i) the Input Hash Unit (IHU): a computational unit handling DMA transfers of input, which constructs “index” and “pointer” hashes for incoming input blocks. (ii) the Input Lookup Unit (ILU): a cache indexed by the index hash value and tagged by the pointer hash, and: (iii) the Computation History Table (CHT): a large table addressed by the pointer hash. The CHT records past accelerator executions. Each CHT entry contains a tuple of: {computation input, computation output}. When fetching the entry, COREx does a full comparison between the fetched input and the input stored in the scratchpad and returns the fetched output in case the inputs match. As efficient table accessing is critical for timing and energy [98], we designed the CHT as a RAM array aggregating multiple memory banks that are orga-
nized in correspondence to two layout parameters: (i) **level of bank interleaving**: dividing the CHT space addressed by the CHT pointer to distinct groups of smaller banks, and (ii): **level of bank parallelism**: distributing a single CHT entry across multiple parallel banks with smaller word size, such that the aggregation of the words read from the parallel banks constitutes the CHT entry.

The execution across all structures is orchestrated by the pipeline controller. All pipeline stages are shown in Figure 4.8. As the figure shows, there are three possible scenarios for the COREx flow: 1️⃣ **hit**: the pointer hash is matched in the ILU, and computation input matches the input of the fetched CHT entry, and the CHT output is used instead of computing the kernel via the accelerator, 2️⃣ **miss**: the pointer is not matched
4.3.3 Input Hashing

The first execution stage is the input hashing stage. This stage is done by the IHU, which incrementally constructs hashes as input blocks are fed to the accelerator. In our experiments, we found that a tree-based bitwise xor is a good fit for incremental construction, and generates a near-uniform distribution. The $N$-bit hash function is described as follows: given the $N$-bit hash value currently stored in the register file, $R_{\text{current}}^{0...N-1}$, an $M$-bit input block, $I^{0...M-1}$ (without loss of generality, we assume: $M = \ell \cdot N$), the hash value calculated in the next cycle, $R_{\text{next}}^{0...N-1}$, is:

$$
R_{\text{next}}^{0} \leftarrow R_{\text{current}}^{0} \oplus I^{0} \oplus I^{N} \oplus I^{2N} \ldots \oplus I^{(\ell-1)N}
$$

$$
R_{\text{next}}^{1} \leftarrow R_{\text{current}}^{1} \oplus I^{1} \oplus I^{N+1} \oplus I^{2N+1} \ldots \oplus I^{(\ell-1)N+1}
$$

$$
\vdots
$$

$$
R_{\text{next}}^{N-1} \leftarrow R_{\text{current}}^{N-1} \oplus I^{N-1} \oplus I^{2N-1} \oplus I^{3N-1} \ldots \oplus I^{M-1}
$$

Figure 4.9a demonstrates the input hashing operation: an “input start” signal is received, marking the beginning of an input DMA sequence. the IHU tracks the DMA
transfers of input blocks and uses the xor-tree to calculate hashes of the input. The IHU concurrently constructs two hashes to use in the lookup stage: (a) the “index hash”, an \( N \)-bit hash used as an index to the ILU, and (b) the “pointer hash”, an \( \tilde{N} \)-bit hash \( (\tilde{N} > N) \) used as a pointer to the CHT, and is stored in the ILU as tag. An “input done” signal received finalizes the incremental hash values, and therefore in \( \Theta \), the IHU initiates a lookup to the ILU via the COREx interconnect.

COREx uses two hash values to achieve both coverage and accuracy. Since the lookup determines whether or not to access the large CHT, its decision is performance and power critical. We use the index hash as an index to the ILU for low latency and power efficiency in case of a miss. The use of a pointer hash as tag increases confidence by reducing the number of “false-hits”, i.e., redundant accesses to the CHT that end up as a mismatch following a full input comparison.

### 4.3.4 Input Lookup

Once the input hashes are constructed, they are used to access the ILU, a set-associative cache indexed by the index hash, and tagged by the pointer hash. The ILU’s data array contains pointers used to access the CHT. Figure 4.9b shows an example of a successful lookup; \( \Phi \) the lookup is initiated by the IHU, which accesses the \( i^{th} \) set using index hash = \( P_{tr_i,2} \); \( \Theta \) the pointer hash is compared with all of the set tags, matching \( P_{tr_i,2} \) in way 2; \( \Theta \) the matched pointer hash, \( P_{tr_i,2} \), is used to access the CHT.

### 4.3.5 Compute-Reuse

The final stage is fetching and reusing results from the CHT. Figure 4.9c shows an example of a 32 banked CHT, in which each CHT entry, i.e., \( \{ \text{kernel\_input}, \text{kernel\_output} \} \) is distributed across four banks, and the CHT address space is divided into eight equally sized spaces. Compute-reuse is done in the following steps: \( \Phi \) a fetch request is sent from the ILU (using the address of \( P_{tr_i,2} \)), matching a group of parallel banks. The fetched CHT
entry is stored in a buffer and forwarded via the interconnect to the accelerator, which, in COREx performs a full input comparison by comparing the kernel input stored in the scratchpad memory, with the \{kernel_input\} from the fetched entry. In, comparison is completed, verifying that the requested input matches the CHT entry, and the result can be used instead of using the accelerator, and is sent via DMA on the SoC interconnect back to the host processor, as the final computation result.

4.3.6 End-to-End Design Flow Example

We demonstrate the end-to-end design flow of the accelerated architecture co-designed with specialized compute-reuse memory layers. This process involves an understanding of the costs of accelerated kernels and underlying memory technology, and an exploration of the memoization tradeoff space to maximize the effectiveness of the memory modules, depending on the optimization goal.

We used the Sum of Absolute Differences (SAD) computation as a case study. In video encoding, SAD is used as a metric for differences between blocks of pixels referred to as: “macroblocks”. Using SAD, the encoder constructs a motion vector for objects in the video. Figure 4.10 provides an auxiliary visualization for the motion vector construction.
**Application Programming:** Memoization requires a high-level understanding of the program. Fortunately, the required reasoning involves defining the memoized kernel and decoupling it from its inputs and outputs, all of which are already done by programmers when writing kernels and programs. Snippet 4.1 demonstrates an implementation of a SAD kernel using an API compatible with heterogeneous programming frameworks such as CUDA or OpenCL. Since the kernel, its inputs, and its outputs are already defined by the programmer, no additional programming effort is needed since memoization correctness is guaranteed by the compiler.

```c
// Definitions: search an 8x8 pixel macroblock
// in a 20x20 pixel search block
#define MX 8
#define MY 8
#define SX 20
#define SY 20
#define dX (SX/2)
#define dY (SY/2)
__kernel__ void SAD(rgb** mb,rgb** sb,int** res) {
  for (int i = 0; i< SX-MX; i++) {
    for (int j = 0; j< SY-MY; j++) {
      res[i][j]=0;
      for (int x = 0; x< MX; x++) {
        for (int y = 0; y< MY; y++) {
          res[i][j]+=abs(sb[i+x][j+y]-mb[x][j]);
        }
      }
    }
  }
}
```

..  

```c
// Main program
for (int x = 0; x< IMAGE_WIDTH; x+=MX) {
  for (int y = 0; y< IMAGE_HEIGHT; y+=MY) {
    id = x*IMAGE_WIDTH + y;
    // dma2DcpyFromHost/Acc(dest,src,x1,x2,y1,y2):
    // DMA copies of a 2D array, src[x1:x2][y1:y2],
    // to a flat array: dest[0..(x2-x1)*(y2-y1)]
    // from the host/accelerator to the accelerator/host
    dma2DcpyFromHost(mb[id],img,x,x+MX,y,y+MY);
    dma2DcpyFromHost(sb[id],ref,x-dX,x+dX,y-dY,y+dY);
    SAD(mb[id],sb[id],acc_res[id]);
    dma2DcpyFromAcc(res[id],acc_res[id],0,SX-MX,0,SX-MX);
  }
}
```

Snippet 4.1: SAD Computation in Video Motion Estimation
Table 4.1: Accelerator Design Sweep Parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explored Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Cycle Time</td>
<td>2ns, 4ns, 6ns, 10ns</td>
</tr>
<tr>
<td>Loop Unrolling Factor</td>
<td>1, 2, 4, 8, 16, 32</td>
</tr>
<tr>
<td>Loop Pipelining</td>
<td>True/False</td>
</tr>
<tr>
<td>Array partitioning (=Mem. Ports)</td>
<td>2, 4, 8, 16, 32, 64</td>
</tr>
</tbody>
</table>

**Accelerator Design:** Since the end of Moore’s law dictates that accelerator gains will be limited by the design points achievable under a fixed transistor budget, we wish to estimate the characteristics of highly-tuned accelerators that are tailored to three different optimization goals: (i) minimal execution time, (ii) minimal energy spent, and: (iii) minimal energy-delay product (EDP). We used Aladdin [151], a performance, power, and area estimation framework for accelerators that operates in a similar fashion to an HLS tool. We tested a range of parameters, detailed in Table 4.1 to find the optimal levels of software pipelining and parallelism, RTL clock cycles, and degrees of memory level parallelism. We explored two accelerator designs: a “BASE” design which refers to an accelerator with SRAM scratchpad memory banks connected to a logic chip with registers and specialized computation lanes, and a “PIM” design which refers to a Resistive-RAM scratchpad packaged with an in-memory logic layer that carries out the arithmetic operations. Figure 4.11 shows all obtained design points (highlighting the per-goal optimal points) and dashed Pareto curves of fixed EDPs.

**Trace Mapping Exploration:** We quantify the level of compute-reuse in the SAD kernel, in order to estimate the effectiveness of memoization tables for the application. We constructed a trace simulator that collects workload input and output traces and characterizes their distribution with respect to different table sizing, from which we derive the table hash width. As input we used traces extracted from ten videos from the YouTube-Faces dataset [187] and classified the table performance according to the number of hits,
misses, and false-hits which were defined in Section 4.3.2. Compute-reuse is expected due to spatial-redundancies in video frame, similar to those discussed in Section 4.2.3.

Figure 4.12 shows the distribution of hit, miss, and false-hit rates for different ILU and CHT hash widths. The figure reflects the following trends: (i) as expected, higher miss rates and false-hit rates are observed for small ILU and CHT sizes, accordingly, and: (ii) the hashing function experiences higher collision rates for hash sizes that are multiples of eight, resulting in the periodic patterns shown in the figure; this is the result of aliasing introduced by the 8-bit RGB channel representation of the video images in the input trace.

**Memory Layer Specialization:** Following the construction of the map of hits, misses, and false hits, we estimate the gains achieved by COREx for each design goal. We integrate the memory layers with the goal-optimized accelerator, and conduct a design sweep for each memory technology and pair of ILU+CHT hash sizes under different CHT layouts. We simulate the tables using Destiny [136], a contemporary emerging memory modeling tool. Figure 4.13 shows the sweeps across the space of possible gains, highlighting the configuration that achieved the best gain. For presentation purposes, we removed non-beneficial configurations (e.g., ILU and CHT sizes that did not achieve speedups). As the
Figure 4.12: SAD Trace Table Exploration. Hit, Miss, and False-Hit Rates as a Function of ILU+CHT Space Sizes.

As the figure reflects, the best speedup for all memory technologies was around $2.7 \times$, while energy and EDP savings depend on the characteristics of each technology’s resistive memory cell and layout. For energy and EDP savings, the beneficial space (i.e., configurations that produce gains) is smaller than the space for runtime. This is due to the use of banking which parallelizes accesses to small aggregated memory modules with low access latency. The use of a large CHT is sub-optimal when optimizing for energy, due to high energy costs of parallel accessing, and static power dissipated by large arrays. Finally, for all design goals, we saw that gains improve as table space increases up to the point of diminishing returns.

## 4.4 Evaluation

In this section, we demonstrate the benefits of using COREx with ReRAMs, PCMs, Race-track memories, and STT-RAMs. We compare COREx against highly-tuned accelerators. We explore accelerators with SRAM-based scratchpads connected to a CMOS-based logic...
Figure 4.13: COREx Gain Space Exploration for SAD. Optimal Gains and Design Points Are Detailed in Each Graph.

chip, similar to previous academic proposals [48,57,106,138,189]. We also examine PIM accelerators that consist of logic cores which are packaged with Resistive-RAM based memory arrays on the same die to reduce data movement costs. We tailor each accelerator to a specific application, and extract input traces to specialize the memory-based tables, following the end-to-end COREx design flow discussed in Section 4.3.6.

4.4.1 Methodology

Evaluated Workloads and Inputs: We outline all evaluated datacenter workloads in Table 4.2. For the video encoding kernels, i.e., “DCT” and “SAD”, we used videos from the YouTube Faces dataset [187], and expect gains from block reuse in videos, as described in Section 4.2.3. “Snappy” is the compression algorithm used by Google, and we used it as an example for traffic compression of Wikipedia query responses. We used 13 million queries generated by a Xapian DB server [95,190] containing abstract dumps published
by the Wikipedia foundation [59]. As we described in Section 4.2.3, web query results tend to hit the same pages, therefore we expect to see recurring results even for different queries, resulting in multiple kernel executions for the same blocks. The growing popularity of graph-centric social networks and content recommendation systems has driven the emergence of graph processing accelerators. We included several graph processing algorithms in our experiments: (i) the single-source shortest-path ("SSSP") algorithm used by map navigation services such as Google Maps; we used the roads of New York as a case study. In this scenario, a datacenter serves navigation requests to find the shortest walking path between two addresses. We used a graph containing five-hundred adjacent streets as accelerator input, as most walking paths are limited to a few blocks. (ii) The Breadth-First Search ("BFS") graph traversal algorithm which we used in the context of product co-purchasing recommendations in online retailer services. In our experiments, we used the Amazon product co-purchasing network taken from data gathered by Leskovec et al. [103]. (iii) The Restricted-Boltzmann Machine ("RBM") is a machine-learning algorithm used by online content services for collaborative filtering, i.e., to detect user-content correlations. We used the Netflix prize dataset [24] to construct an accelerator for a movie recommendation system.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Domain</th>
<th>Use-Case</th>
<th>Kernel Source</th>
<th>Input Source and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSSP (&quot;SSP&quot;)</td>
<td>Graph Processing</td>
<td>Maps Service: Shortest Walking Route</td>
<td>Internal</td>
<td>DIMACS, Graphs of NYC Streets [53], 10 Million Zipfian Transactions, 264K Nodes, 733K Edges.</td>
</tr>
<tr>
<td>BFS</td>
<td>Graph Processing</td>
<td>Online Retail</td>
<td>MachSuite [140]</td>
<td>Amazon Co-Purchasing Network [103], 10 Million Zipfian Transactions, 403K Nodes, 3M Edges.</td>
</tr>
</tbody>
</table>
**Zipfian traffic**: Datacenters and web-servers are designed to satisfy biased traffic. More specifically, traffic has been shown to abide by Zipf’s Law; a statistical linguistic law, formulated in 1929 [207] stating that given a corpus of words (i.e., a large dictionary), words are distributed in an inversely proportional manner to their popularity, i.e., the \( i^{th} \) most common word is distributed proportionally to \( i^{-1} \). Interestingly, Zipf’s Law was shown to apply to internet traffic [7,33], and it was empirically shown to be the key component for enterprise traffic compression [13]. Therefore, as some datacenter requests are more popular than others, some computations have high repetition rates, and this repetition might manifest as recurrent (and redundant) accelerator computations. For “SSSP”, “BFS”, and “RBM”, we generated traces that follow a Zipfian distribution with 75%, 90%, and 95% skew rates. These skew rates are conservative compared to the rates used by recent studies [46,105,106].

**Accelerator, COREx Tables, and Interconnection Modeling Framework**: For each workload, we ran a design sweep for the accelerator core using the Aladdin modeling framework [151] and the Destiny memory modeling tool [136] to simulate both scratchpad and the ReRAMs for PIM-based accelerators. We extracted the best results for each design goal. We also used Destiny to model the ILU and CHT memory layers in COREx as ReRAM, PCM, STT-RAM, and Racetrack memories. We used these technologies as it is likely for some of them to scale post the end of Moore’s Law. We used heavy memory banking and block-level parallelism to achieve low latency and low dynamic energy accesses to the large CHT storage layer. Since the hashing scheme is coupled with the table organization and banking mapping, it is bank aware, which enables the selection of the appropriate bank via muxing done outside the CHT (i.e., lookup phase) to further reduce energy. Since Aladdin models 45nm CMOS logic and scratchpads, we use 45nm memories for fair comparison. We set interconnection energy and latency following recent network-on-chip (NoC) studies [21,114,132]. For each workload, we swept across both traditional

---

1 TailBench is a client-server benchmark suite, in which the server provides the computational services for the client; therefore, we focused our evaluation on the server side only.
Table 4.3: Kernels and Optimal Design Sweep Points.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>DCT [µs]</th>
<th>SAD</th>
<th>SNP</th>
<th>SSP</th>
<th>RBM</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>7.32</td>
<td>5.85</td>
<td>0.88</td>
<td>1.95</td>
<td>50.15</td>
<td>2.69</td>
</tr>
<tr>
<td>Energy</td>
<td>8.98</td>
<td>6.16</td>
<td>10.17</td>
<td>2.98</td>
<td>1.5K</td>
<td>15.03</td>
</tr>
<tr>
<td>EDP</td>
<td>396.88</td>
<td>148.75</td>
<td>8.99</td>
<td>8.05</td>
<td>78K</td>
<td>64.06</td>
</tr>
</tbody>
</table>

Table 4.4: General System Parameters.

<table>
<thead>
<tr>
<th>COREx Interconnect</th>
<th>Bank Muxing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flit-Size</td>
<td>Latency</td>
</tr>
<tr>
<td>64-bit</td>
<td>1ns</td>
</tr>
</tbody>
</table>

and PIM accelerators and picked the optimal points as the baseline accelerator. Table 4.3 summarizes the optimal accelerator design sweep points. The parameters for interconnect and muxing are shown in Table 4.4.

**Evaluated Configurations:** In our evaluation we use the tuned accelerators as the baseline for comparison, and tune COREx for three optimization goals: Runtime-OPT is the configuration that achieves the best speedup compared to the runtime-optimized accelerator; similarly, Energy-OPT and EDP-OPT are the configurations that save the most in energy and energy-delay-product, for the Energy-OPT and EDP-OPT accelerators, respectively. We explore different memory layer implementation alternatives for the ILU and CHT using emerging memory models that are used by researchers: ReRAMs [31,38,201], PCMs [102,147], STT-RAMs [100,198], and Domain-Wall Memories (DWMs), commonly known as Racetrack memories [177,199]. We derive different configuration design points based on the optimization objectives and the relations between the given workload and underlying memory technology. Table 4.5 shows the optimal ILU and CHT configurations for each memory and optimization goal. For presentation purposes, we present the optimal table configurations for a subset of workloads. We used a 4-way associative organization for ILU, since we found that it provided a desirable hit-rate to cost tradeoff.
Table 4.5: Optimal Table Configurations for a Subset of Workloads, Following Memory-Layer Specialization; CHT Total Size and Layout ($N \times M = N$ Interleaved $\times M$ Parallel Banks), CHT Bank Latency, and ILU Size. ILUs are 4-Way Set-Associative.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Optimization Target</th>
<th>CHT</th>
<th>PCM</th>
<th>ILU</th>
<th>CHT</th>
<th>Racetrack</th>
<th>ILU</th>
<th>CHT</th>
<th>ReRAM</th>
<th>ILU</th>
<th>STT-RAM</th>
<th>ILU</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCT</td>
<td>Runtime</td>
<td>4GB:2×64(12ns)</td>
<td>524KB</td>
<td>4GB:2×64(12ns)</td>
<td>524KB</td>
<td>4GB:2×64(12ns)</td>
<td>524KB</td>
<td>4GB:2×64(12ns)</td>
<td>524KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>16MB:2×1(374ns)</td>
<td>524KB</td>
<td>2MB:2×1(374ns)</td>
<td>524KB</td>
<td>16MB:2×1(374ns)</td>
<td>524KB</td>
<td>16MB:2×1(374ns)</td>
<td>524KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDP</td>
<td>536MB:2×8(374ns)</td>
<td>524KB</td>
<td>134MB:2×16(193ns)</td>
<td>262KB</td>
<td>536MB:2×8(374ns)</td>
<td>524KB</td>
<td>536MB:2×16(321ns)</td>
<td>524KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNP</td>
<td>Runtime</td>
<td>274GB:2×512(14ns)</td>
<td>8MB</td>
<td>11TB:2×512(13ns)</td>
<td>8MB</td>
<td>11TB:2×512(10ns)</td>
<td>8MB</td>
<td>11TB:2×512(6ns)</td>
<td>8MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>1TB:2×32(374ns)</td>
<td>8MB</td>
<td>No Gains</td>
<td>8GB:2×256(9ns)</td>
<td>8MB</td>
<td>1GB:2×32(321ns)</td>
<td>4MB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDP</td>
<td>4GB:2×512(102ns)</td>
<td>1MB</td>
<td>1GB:2×512(50ns)</td>
<td>131KB</td>
<td>34GB:2×512(20ns)</td>
<td>8MB</td>
<td>2GB:2×512(82ns)</td>
<td>1MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBM.75%</td>
<td>Runtime</td>
<td>2TB:16×9K(11ns)</td>
<td>134MB</td>
<td>1TB:2×37K(2ns)</td>
<td>134MB</td>
<td>1TB:2×37K(1ns)</td>
<td>134MB</td>
<td>1TB:2×37K(2ns)</td>
<td>134MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDP</td>
<td>19GB:2×293(374ns)</td>
<td>2MB</td>
<td>1GB:2×73(764ns)</td>
<td>4KB</td>
<td>19GB:2×36(7ns)</td>
<td>2MB</td>
<td>19GB:2×146(1µs)</td>
<td>2MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBM.95%</td>
<td>Runtime</td>
<td>1TB:32×18K(11ns)</td>
<td>8MB</td>
<td>629GB:2×37K(1ns)</td>
<td>8MB</td>
<td>629GB:2×37K(1ns)</td>
<td>8MB</td>
<td>629GB:2×37K(1ns)</td>
<td>8MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>4GB:2×36(1µs)</td>
<td>8MB</td>
<td>76MB:2×73(64ns)</td>
<td>65KB</td>
<td>1GB:2×586(13ns)</td>
<td>67MB</td>
<td>2GB:2×36(1µs)</td>
<td>16MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDP</td>
<td>19GB:2×73(1µs)</td>
<td>65KB</td>
<td>1GB:2×73(64ns)</td>
<td>4KB</td>
<td>19GB:2×36(3ns)</td>
<td>2MB</td>
<td>19GB:2×146(1µs)</td>
<td>65KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS.75%</td>
<td>Runtime</td>
<td>8TB:16×2K(14ns)</td>
<td>16MB</td>
<td>8TB:4×2K(13ns)</td>
<td>16MB</td>
<td>8TB:4×2K(10ns)</td>
<td>16MB</td>
<td>8TB:4×2K(16ns)</td>
<td>16MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td>No Gains</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDP</td>
<td>1GB:2×259(374ns)</td>
<td>131KB</td>
<td>543MB:2×2(13ns)</td>
<td>16KB</td>
<td>17GB:2×2K(7ns)</td>
<td>524KB</td>
<td>1GB:2×64(321ns)</td>
<td>131KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS.95%</td>
<td>Runtime</td>
<td>8TB:4×2K(20ns)</td>
<td>8MB</td>
<td>8TB:4×2K(13ns)</td>
<td>8MB</td>
<td>8TB:4×2K(10ns)</td>
<td>8MB</td>
<td>8TB:4×2K(16ns)</td>
<td>8MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>536MB:2×16(374ns)</td>
<td>524KB</td>
<td>33MB:2×32(50ns)</td>
<td>65KB</td>
<td>543MB:2×586(13ns)</td>
<td>16MB</td>
<td>134MB:2×8(321ns)</td>
<td>131KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EDP</td>
<td>1GB:2×129(102ns)</td>
<td>262KB</td>
<td>543MB:2×2(13ns)</td>
<td>16KB</td>
<td>17GB:2×2K(7ns)</td>
<td>524KB</td>
<td>1GB:2×64(321ns)</td>
<td>131KB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4.2 Results

Figure 4.14 shows the gains in runtime, energy, and EDP for the workloads evaluated using the design points in Table 4.5.

Results Summary: Compared to the runtime optimized accelerator, COREx achieves an average speedup of 6.0× for PCM, and 6.4× for all other memory technologies. In energetic terms, COREx saves an average of 22%, 27%, 48%, and 50%, for PCM, Racetrack, Resistive-RAM, and STT-RAM, respectively. The EDP savings were 60%, 50%, 68%, and 62%, for COREx configurations using PCM, Racetrack, Resistive-RAM, and STT-RAM, respectively.

Behavior under Different Objectives and Skews: As reflected by Figure 4.14 and Table 4.5, the increase in Zipfian traffic skew achieves higher speedups, but does not result in a change in the table size in Runtime-OPT. We found that the reason for this is that the workload configuration for these workloads was set to the largest CHT (determined by workload input and output sizes), and therefore achieved the highest hit rate. Furthermore, we found that CHT hit rate had a higher impact on the timely performance of COREx than...
lookup costs (i.e., Muxing, ILU, NoC, and CHT bank latencies). When energy was the goal, we saw that table sizes grew as skew increases. The reason for this behavior stems from the fact that for higher skews, the marginal leakage and access costs of increasing the table size were smaller than the energy gained from increasing the hit rate and using the tables instead of the accelerator.
**Memory Technology Implications:** In our experiments, we saw that all memory technologies achieved roughly the same speedups with minor differences, this is the result of COREx’s parallel CHT banking structure, enabling shorter table access latencies compared to accelerator computation speeds, while the choice of underlying memory technology plays a less significant role. In general, when runtime is the primary objective, optimal CHT sizes are large, and are roughly the same size for all memory technologies, given the same workload. For PCM, we see that in three benchmarks there were no configurations that achieved energy gains, but we saw that gains were achieved for the same benchmarks with higher skews. We saw that the reason for this is the high dynamic energy costs dominating PCMs compared to the other technologies explored. We also saw that STT-RAMs and ReRAMs achieved superior and similar energy and EDP rates across most of the workloads.

**Energetic Breakdowns:** Figure 4.15 shows the energetic breakdown of different COREx components for the Energy-OPT configurations, compared to the baseline accelerator-only configuration. We present the energy spent by the computing accelerator core, the leakage energy from CHT arrays, the dynamic CHT access energy (reads/updates), and all other components (ILU leakage and dynamic energy, the scratchpad memory (SPM) reads for full-input comparison and NoC interconnect traffic). As expected, for workloads that have large traffic skews, tables can save the energy spent in accelerator core computations and reduce overall energy spent. The energy costs of COREx components are dominated by the accelerator core, followed by the rates of leakage energy of the CHT. In contrast to the other memory technologies, current PCM models reflect relatively high write energy and latency, making CHT updates expensive. As reflected by Table 4.5, since typical ILU storage size is orders-of-magnitude smaller than CHT size, static energy rates are negligible. This is also true for the energy spent by other components (e.g., NoCs and Scratchpad reads).
Figure 4.15: Energetic Breakdown for Energy-OPT Schemes.

**An Outlook on Compute-Reuse for Accelerators:** As the experiments conducted show, existing non-volatile memory technologies can deliver performance and energy returns for accelerated architectures, although they depend on traffic skews and other sources of computational redundancies for beneficial memoization. As discussed in Section 4.2.1, since current projections show a steady increase in storage density, **architectures such as COREx which leverage storage as their primary source of gain will become more appealing.** We speculate that future technology roadmaps will enable the construction of denser tables with smaller arrays and expand the applicable table space for compute-reuse architectures, thereby delivering the same hit rates for smaller costs, or by improving accelerator returns for lower hit-rates.

### 4.5 Conclusions

We have presented COREx, a storage driven compute-reuse architecture for future data-center accelerators. We discussed the implications of the end of transistor scaling in accelerators, while presenting still-scaling storage as means to make future improvement in computation capabilities possible. We integrated contemporary emerging memory technologies to construct COREx in order to compensate for non-scaling CMOS technology by transforming transistor-based computations to table storage which continues to scale. We presented a detailed overview of the COREx architecture, and apply it to accelerated...
kernels to improve runtime, energy, and EDP. Beyond the presented gains, the concept of transforming problems to the still-scaling storage domain shows promise for future architectures. We believe that these gains will improve as future memory technologies keep scaling, paving the way for storage-centric architectures.
Chapter 5

REDS: A REuse-Driven Specialization Architecture for Accelerator-less Clouds

Accelerator-based specialization became a prominent means to ameliorate the gap between the growing computation demands of cloud services and the stagnation in processing chip capabilities due to the slowdown of CMOS scaling. While delivering superior efficiency rates compared to general-purpose processors, not all cloud domains can benefit from accelerators. Accelerators take a long time to deploy due to the process of defining special-purpose interfaces, exploring hardware-software co-designs, and possibly developing an entirely new chip. In the period of time between the identification of new computation needs and the deployment of the new hardware accelerators, domains are left accelerator-less.

We present REDS, an architecture that employs REuse-Driven Specialization for accelerator-less cloud domains. An accelerator-less domain consists of emerging or rapidly-changing computation patterns that are not supported by existing accelerator hardware and is therefore solely dependent on inefficient general-purpose processors. REDS leverages the commonality of cloud traffic and function-centric nature of modern clouds to integrate a memory layer with the general-purpose processors. The added layer stores
the outcomes of previous functions and reuses them in case of recurring computations. By co-designing the networking, compute, and memory layers, REDS extends programmable processing with a specialized memory layer to deliver near-accelerator compute efficiency rates.

We evaluate REDS as an extension to an x86 processor-based cloud consisting of 4096 compute nodes. Our results show that, when compared to highly-tuned fixed-function accelerators, REDS can speed up execution time by an average 20×, or deliver efficient execution with an average of about 3× in runtime energy overhead.

5.1 Introduction

Datacenters and cloud services are the backbone of the world’s computation, catering to the needs of billions of people on a daily basis. However, computational demands are constantly rising and pose challenges to datacenter designers and owners due to high energy rates [154]. One of the prominent solutions to alleviate the growing computational stress on cloud datacenter infrastructure is the integration of specialized “accelerator” chips such as FPGAs [12,60,137], and ASICs [89,113]. Accelerators compromise on programmability and target specific domains to deliver orders-of-magnitude improvements in computation efficiency.

Future Clouds Challenge Accelerator-Based Specialization: The reliance on rigid hardware structures and interfaces along with compromised programmability make accelerator-based specialization risky for future clouds. In specific, with the recent popularity of serverless computing, cloud workloads shift towards flexible, rapidly-changing, and function-centric computations [67,133]. In modern cloud environments, it is easy to add or remove functions without affecting other modules. Functions can scale up and down instantaneously and are provisioned independently [61]. Unfortunately, these properties also make many future cloud workloads unfit for accelerator-based specializa-
tion. Long chip development times make ASIC-based acceleration impossible for fast deployment [127], and compromised programmability limits the support responding to new market application demands using fixed hardware and optimized interfaces [126]. As recently surveyed, most cloud developers do not employ accelerators for their computations. 99% of cloud programs (serverless and traditional) written in high-level languages run on general-purpose x86-based processors [87]. As trends continue, an emerging question is: “how can clouds attain efficient computation rates, even when an accelerator is not present?”

**Future Clouds Pose an Opportunity for Computation Reuse-Driven Specialization:** Function-centric clouds, such as serverless Function-as-a-service (FaaS) platforms, are designed to run independently-orchestrated, generally-stateless, functions that have well-defined interfaces with decoupled input, output, and computation. It is, therefore, possible to establish a deterministic relation between a computation’s input and output, making modern cloud platforms a natural target for function-level computation-reuse specialization via memoization [118]. Furthermore, cloud services possess a high degree of computation commonality, as different cloud nodes process similar popular computations. Finally, the skew of data values being accessed and computed on in cloud services such as media [123] and social networks [121] make the case for profitable memoization due to high computation reuse rates.

We present REDS, an architecture that employs REuse-Driven Specialization for **accelerator-less** clouds. REDS extends the general-purpose processing fabric with a specialized memory hierarchy that stores the results from previously-computed functions. When a previously-computed input is requested, the stored result is fetched from the table instead of being redundantly recomputed via the processor. Figure 5.1 shows the integration of a many-accelerator SoC with a memory hierarchy for reuse-driven specialization of emerging and volatile cloud domains. Since emerging domains require the design of an entirely new accelerator architecture, and volatile domains incur frequent
functionality changes that are not supported by existing accelerator hardware, they cannot execute on the SoC’s designed accelerators. The key motivations and ideas for REDS are as follows. (i) Rapidly-evolving cloud development and workloads make it difficult to use hardware accelerators for all applications. (ii) In contrast to accelerator-based specialization, computation-reuse does not depend on the inner computation characteristics (e.g., operand types, potential ILP, etc.) and is therefore a versatile and microarchitecturally-agnostic form of specialization. (iii) Content popularity, along with the well-defined input-compute-output flows and stateless nature of function-centric clouds make reuse-driven specialization beneficial. By shifting computations from inefficient general-purpose processors to a reuse hierarchy tailored to computation needs, REDS can achieve near-accelerator execution efficiency while maintaining flexibility, as qualitatively outlined in Table 5.1.

\[\text{Table 5.1}\]

\(^1\text{While GPPs are fully-programmable, not all computations are amenable for reuse-driven specialization, as discussed later.}\]
Table 5.1: Qualitative Comparison: GPPs, Programmable Accelerators (e.g., FPGAs), DSAs, and GPPs with Reuse-Driven Specialization.

<table>
<thead>
<tr>
<th></th>
<th>Flexibility</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>General-Purpose Processors (GPPs)</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Programmable Accelerators</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Domain-Specific Accelerators (DSAs)</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>GPPs+REDS (This Work)</td>
<td>High</td>
<td>Workload Dependent</td>
</tr>
</tbody>
</table>

The contributions of this chapter are as follows:

- We present a discussion of accelerator-less workloads and how they can be specialized using computation-reuse.
- We develop the REDS API and end-to-end flow as a processor-network-memory co-design, and explore the tradeoffs of a cloud-scale computation-reuse space.
- We evaluate REDS as an extension to an x86-based processing fabric on a 4096 compute nodes cloud, exploring 100,000 design alternatives on a set of accelerator cloud functions. We show that REDS can achieve an average of $21 \times$ in speedup or execute with an average energy overhead of only $1.4 \times$ compared to highly-tuned fixed-function accelerators, if they were to be built.

5.2 Background and Motivation

We discuss the recent shifts in cloud service applications, as well as the trends that motivate reuse-driven specialization.

5.2.1 A Shift in Cloud Computing Paradigms

The growing popularity of new cloud paradigms, such as serverless computing and microservices, change the way cloud programming environments are written and how data-
center infrastructure is managed. These new paradigms offer decoupled blocks of computation with automated scalability, reduced overhead of resource provisioning, and no idle computation costs [43]. These features are the result of a rising need for cloud flexibility, programmability, and ease-of-deployment. More specifically, major cloud vendor now offer serverless FaaS platforms such as AWS Lambda [148], Google Cloud Functions [70], IBM Cloud Functions [83], and Microsoft Azure Functions [119].

These emerging cloud platforms signify the shift towards new computing models that consist of loosely-coupled, mostly non-stateful cloud functions with well-defined interfaces. Most importantly, the combination of well-defined function-centric environments and high computation redundancy due to skewed cloud traffic [7,33,63] makes future cloud functions a natural target for computation reuse.

5.2.2 Accelerator-less Cloud Domains

Cloud services run on commercial datacenters that are computationally-intensive, and demand high amounts of power and energy. The energy consumption of U.S datacenters alone is projected to rise to an annual rate of 73 billion kWh by 2020 [154]. The majority of a datacenter’s power and energy is spent by its processors. For example, a 2017 study conducted by Google’s datacenter team, showed that 61% of its peak datacenter power is spent by CPUs [22]. While accelerators improve computational efficiency by orders-of-magnitude, they require domains to be mature and stable [64]. Inflexible accelerator hardware could become obsolete or could impede algorithmic innovation in volatile domains [126]. For emerging domains, the creation of a new accelerated architecture is a time-consuming process that requires developing, prototyping, and defining new hardware/software interfaces. We show real-world examples of such accelerator-less domains.

Emerging Domains. Roadmaps to DNN Cloud Acceleration: The rising popularity of deep learning produced a plethora of compute-intensive deep neural network (DNN) workloads that stress existing datacenter infrastructure. In 2013, Google’s researchers pro-
jected that the computation demands of speech recognition alone would force a doubling of the number of CPU-based servers in their datacenters [89]. This motivated the development of an ASIC chip called the Tensor Processing Unit (TPU) that was aiming at efficient DNN execution, and it was announced in mid-2016, following fifteen months of ASIC development, verification and deployment [88]. Similarly, in 2016, Microsoft’s engineers began developing project Brainwave [60] for FPGA-accelerated DNNs in the cloud, using their existing FPGA infrastructure for cloud microservices [37]. According to the Azure blog, about a year later Microsoft deployed the first boards of the Brainwave neural processing unit (NPU) [120]. In both cases, there was a period of about a year from the identification of the computation need to the deployment of first accelerators. During this period, DNN computation was a popular, compute-intensive domain that was executed in an accelerator-less cloud setting.

**Volatile Domains. Smart Speakers Functionality:** The adoption of smart speakers, such as Amazon Alexa and Google Home, grew by 40% to 66.4 Million [182], and it is driven by the flexibility of serverless FaaS cloud computing service. One of the key advantages of serverless FaaS clouds is that they provide businesses with immediate value due to relatively short deployment times of new ideas and emerging technologies [40]. This flexibility also yields a rapid growth in the diversity of cloud functions. Figure 5.2 shows the timely evolution of cloud functions for Amazon Alexa and Google home deployed in the United States [180,181]. As seen in the figure, dozens of new functions are added on a daily basis, and they necessitate *new interfaces and protocols*, for example: connecting to new edge devices, introducing new commands to a smart home, etc. This degree of volatility makes smart speaker cloud-serving a domain that is likely to remain accelerator-less.

---

89
5.2.3 A Primer on Reuse-Driven Specialization

We explore the use of memoization to leverage computation-reuse as an additional dimension in datacenter specialization. Memoization is a technique first termed by Michie in 1968, as a form of a machine that learns from past experiences [118]. In memoization, the results of previous computations are stored in tables and are indexed by the corresponding inputs (i.e., the inputs that generated the computation results). When a previously computed input is encountered, the corresponding result is fetched from the table instead of being redundantly recomputed. By learning past computations, the computing system amortizes computation costs and becomes more specialized for a specific computation and/or workload, trading processing for memory accesses. Figure 5.3 motivates the rationale for this processing-memory tradeoff. The figure plots the runtimes and runtime energies of a K-nearest neighbors (KNN) clustering function for a fixed-function accelerator, and an out-of-order x86 CPU with 1GB of DRAM-based reuse-memory for different reuse hit rates. As the figure shows, compared to an accelerator, the CPU has a runtime overhead of 1.8× and an energy overhead of about 100×, but for a reuse hit rate of 95% it can speed up execution by a rate of about 11× while reaching a reduced energy overhead of about 5.5×,
therefore achieving near-accelerator efficiency rates, without incurring the overheads of building an accelerator.

In contrast to domain-specific accelerators, computation-reuse is an architecturally-agnostic form of specialization. It does not assume any computation-specific characteristics such as parallelism degree, inner-data dependencies, or non-trivial functionality (e.g., ISA support for Tangent/ReLU activation functions in CNNs [60,89]). Memoization solely depends on the most fundamental invariants of a computation, which is the relation between its input and its output. From a computational perspective, memoization requires the minimal information needed to know about a given problem meaning: (i) what is the problem (i.e., the computed function) and (ii) what is its input. Reuse-driven specialization requires a data-driven examination of the targeted function to achieve profitability, as a function of table hit rates, $P_{HIT}$, costs of input lookup and matching, $C_{LOOKUP}$, storing past computations, $C_{STORE}$ and fetching them from the table, $C_{FETCH}$, compared to the original computation costs of the function, $C_{FUNCTION}$, as shown in the following equations:
\[ C_{\text{FUNCTION}} \geq C_{\text{LOOKUP}} + (P_{\text{HIT}} \cdot C_{\text{HIT}}) + (1 - P_{\text{HIT}}) \cdot C_{\text{MISS}} \]

\[ C_{\text{MISS}} = \underbrace{C_{\text{FUNCTION}}} + \underbrace{C_{\text{STORE}}} \]

\[ \text{Fallback to Original Computation} \quad \text{Update Table With New Result} \]

\[ C_{\text{HIT}} = \underbrace{C_{\text{FETCH}}} \]

\[ \text{Read and Send Stored Results} \]

\[ \Downarrow \]

\[ C_{\text{FUNCTION}} \geq \frac{C_{\text{LOOKUP}} + (1 - P_{\text{HIT}}) \cdot C_{\text{STORE}}}{P_{\text{HIT}}} + C_{\text{FETCH}}; 1 \geq P_{\text{HIT}} > 0 \]

We conclude that to reach profitability for a given function with a computation cost of \( C_{\text{FUNCTION}} \), reuse-driven specialization must strive to achieve the following (often conflicting) goals; the maximization of hit rates (\( P_{\text{HIT}} \uparrow \)) and minimization of costs of computation storing (\( C_{\text{STORE}} \downarrow \)), lookup and matching (\( C_{\text{LOOKUP}} \downarrow \)), and fetching results (\( C_{\text{FETCH}} \downarrow \)). For a given computation or workload, these variables determine the explored optimization space of designed architectures.
5.3 Reuse-Driven Specialization: Opportunities and Challenges

We present the motivating trends and guidelines for the devising of a cloud architecture with reuse-driven specialization.

5.3.1 Cloud Power Laws

Cloud traffic possess highly-skewed distributions rates, set by the popularity laws that govern user access patterns. Cloud users are orders-of-magnitude more likely to access popular cloud content than unpopular content. Figure 5.4a shows the empirical rates of playlist popularity of the music streaming service Spotify extracted from a dataset released as part of a recommendation systems challenge [160]. Figure 5.4b shows the distribution of keys with respect to the number of requests for five real-world applications. The key distribution data was synthesized from a characterization study on Facebook’s production machines [19]. Cloud content popularity has been shown to abide by Zipf’s law [7,33], an empirical law originally formulated for linguistic distributions [208]. Zipf’s law has been later used to evaluate and benchmark cloud serving systems [46,58,63,105,106]. In a Zipfian distribution, content is ranked in accordance to its popularity. The occurrence probability of the $i^{th}$ most popular item is inversely proportional to its rank, as dictated by a bias traffic parameter, $\alpha$, i.e., $\text{Prob.}(Item_i) \propto i^{-\alpha}$.

5.3.2 Caching Computations at Cloud Scale

Cloud services process high rates of recurring computations, e.g., search engines fetch and process similar popular pages, streaming services decode the same videos, etc. We employ reuse-driven specialization using a distributed storage hierarchy to cache previously-computed functions and extend the general-purpose processing fabric. The cache hierarchy records past computations of a function, $\Psi$, identified by $f_{id}$. Given a sequence of past
inputs $IN_1, IN_2, ..., IN_\ell$, with corresponding outputs, $\Psi(IN_1), \Psi(IN_2), ..., \Psi(IN_\ell)$, the cache stores entries of the form $<f_{id}, IN_k, \Psi(IN_k)>$.

**Domain Volatility:** Like other forms of specialization, computation caching benefits from workload stability. Unfortunately, for volatile domains, the targeted computation can change rapidly. Changes can reduce the effectiveness of caches by possibly rendering previously-stored computations obsolete. In this section, we account for the loss of cache effectiveness by assigning an *epoch* that signifies the number of consecutive function calls before a computation change takes place. A new epoch signifies a change that was made in a way that makes old computation results unusable\(^2\).

**Distribution Tails:** While caches are natural means to exploit skewed distributions, computed content is typically diverse, and the possible computation space is unlikely to map to any practically-sized cache hierarchy. As a result, the long “tail” (i.e., low-ranked) content might conflict with and evict popular content, resulting in reduced hit rates and energy wasted by redundant storing of transient (i.e., non-

\(^2\)This is a conservative worst-case approach, since not every computation update would nullify all previous computations unusable.
5.3.3 Filtering and Commonality at Cloud Scale

**Experimental Setup:** We explore filtering and commonality on a 4096 machine cloud with a 128GB caching hierarchy. Each of the sharing nodes contains a portion of the cache hierarchy as four-way set-associative. Each cache line stores 16KB objects, i.e., a total of two million keys are cached in a given time. We analyzed the caching of computations that follow a Zipfian traffic. We analyzed traffic traces of four million requests with different degrees of Zipfian parameter bias: \( \alpha = 1.3, 1.6, 1.9 \), that are typical traffic skew parameters used by previous studies (e.g., \( \alpha = 1.6 \) is used in Linkbench, a benchmark suite that generates social graph access patterns based on production data from Facebook [16].)

The long tail problem mentioned in Section 5.3.2 necessitates mechanisms to improve the reuse-storage beyond basic caching algorithms. We explore the addition of a filtering mechanism to prevent cache pollution by the insertion of unpopular content. Algorithm 1 shows the basic flow of reuse-driven specialization with “second chance” filtering \(^3\). The goal of this technique is to reduce the number of evictions of popular content by unpopular content and reduce the pollution by transient content (i.e., unusable computation outputs that are cached, but not used before eviction). To motivate the use of filtering, and determine an appropriate filter size, we analyze the reuse distance of Zipfian traffic. The reuse distance is the average number of computations between two consecutive instances of a given key. Figure 5.5 shows the reuse distance for the 100 most popular keys. The figure implies that storing a group of a hundred keys would be sufficient to capture keys with reuse distances of at least a thousand (less popular keys have a higher reuse distance), therefore filtering out keys outside the hundred key group would result in fraction of a percent loss in hit rate. We analyze non-filtered cache and a filtered a cache with a filter of 128 keys.

Figure 5.6 shows an analysis of stable traces as well as volatile traces with an epoch of 1000 (i.e., stored computations are nullified every 1000 computations). The figure reflects

---

\(^3\)For presentation purposes, we assume one-to-one correspondence between the key space and the function+input space, i.e., no need for an additional function+input lookup following the key lookup.
Algorithm 1 Second Chance Filtering

```plaintext
key ← HASH(func,input)
if key in Cache // Lookup Hit. Use cached compute output
    output ← Cache[key]
else  // Lookup Miss
    output ← COMPUTE(func,input) // Fallback to original compute
    if key ∉ Key_Filter // 1st Chance. Put in key filter
        Key_Filter.add(key)
    else  // 2nd Chance. Store in cache
        Cache[key] ← output
        Key_Filter.remove(key)
return output
```

the following insights. (i) Filtering increased hit rates for stable workloads, but reduced hit rates for volatile workloads, as shown in Figure 5.6a. The reason for this behavior is that for stable workloads selective insertion prevents the eviction of popular content by unpopular content, but since it also extends the cache warmup period (need to hit the filter before inserting), which is a longer fraction of workload times for rapidly-changing domains. It is possible to overcome this issue by implementing hardware support for warmup periods (e.g., following a change in the targeted computation) during which filtering is disabled. (ii) Filtering significantly reduces cache pollution for both stable and volatile workloads, as shown in Figure 5.6b. We consider pollution as computations that are stored in the cache, but are evicted prior to being reused. The reason that there are still a few such stores is due to hash collisions in the cache between popular keys evicting each other before they are
used by the cache. As computation inputs and outputs are on the orders of KBs to MBs, filtering can easily save GBs of redundant memory writes in the datacenter.

recurring) computations.

As large-scale cloud services run on datacenters, and are typically deployed on thousands of machines, an important aspect of highly-distributed hierarchies is datacenter commonality. Different machines are likely to serve similar content, and therefore can benefit from commonality-driven sharing across the compute-reuse caching hierarchy. Furthermore, due to traffic bias, “hot” content is likely to be redundantly stored across multiple caches, reducing the effective cache size. To estimate the impact of commonality at the datacenter-scale, we explore how the 128GB cache hierarchy performs, as it is distributed across the 4096 nodes. We explored different sharing degrees, a sharing degree of 1 means each node has its own private 32MB cache. Similarly, a sharing degree of 4096 means all nodes share a single four-way associative 128GB cache. Figure 5.7 shows the cumulative distribution of cache misses with respect to key ranks (i.e., the number of misses for keys that are ranked higher than \( \text{rank} = r \), which is represented by the x-axis.). We analyze different traffic skews, and for different cache sharing degrees across the datacenter. To provide further intuition on the distribution of misses we analyze the behavior of the “fifty
percentile” miss rank (i.e., the median key rank \( r \) for which the number of misses for keys ranked higher than \( r \) equals to the number of misses for keys ranked lower than \( r \)). In Figure 5.8, we quantify the behavior of sharing with respect to different levels of domain volatility (i.e., different epoch lengths).

Figures 5.7+ 5.8 reflect the following. (i) Commonality improves cache effectiveness. As Figures 5.7a,5.7b,5.7c show, for all traffic bias rates, the number of overall misses drops for higher commonality traces. This is caused by the increase in effective cache sizes due to irredundant storage of hot keys that improves caching efficacy. Furthermore, the increase of commonality shifts the fifty percentile marks to the right, i.e., towards the lower ranked keys, suggesting that the relative portion of popular key misses is smaller. The shift is caused by the increase in storage size which reduces the probability of conflict misses caused by the eviction of popular keys by less popular ones. (ii) Commonality ameliorates the impact of domain volatility on caches. As Figure 5.8 shows, while volatile computations (i.e., short epochs) impact the overall miss rate, commonality can be used to reduce the loss of locality. Miss rate patterns were less noisy for the higher commonality traces, suggesting that the impact of volatility is moderate. Furthermore, as commonality increases, the volatile traces (i.e., epoch=500) are getting closer more-stable traces (i.e., epoch=2000). The reason is that for transient cases the impact of cold-start misses is greater. For keys that are uniformly distributed across the system (higher ranked keys), an \( N \)-way shared cache saves \( N - 1 \) cold start misses compared a private cache, since the shared key is stored and warms up the cache only once. Therefore, sharing can make up for loss of caching effectiveness due to warmups in a filtered caching environment.

We conclude the discussion on filtering and commonality in terms of factors for memoization presented in Section 5.2.3, and summarize our findings in Table 5.2. Filtering reduces the costs of stores (\( C_{STORE} \)) since it amortizes the number of polluting cache writes, it does not change the cost of computation lookup (\( C_{LOOKUP} \)) or fetching (\( C_{FETCH} \)) for hits (although it can slightly increase the lookup cost for misses and writes, that are not in the
critical path for performance). As discussed earlier, the impact of filtering on hit rate (i.e., \( P_{HIT} \)) depends on workload volatility. We saw that commonality improves hit rates. Commonality’s impact on the cost of storage is workload and topology dependent and requires further exploration. It can either increase costs (due to congestion on a shared cache) or it can also decrease costs (a shared key is stored only on the shared cache instead of multiple private caches). Lookup and fetch costs generally rise, but rising costs can be mitigated by sharding the hash keys.

### 5.4 Architecture

REDS deploys a distributed memory hierarchy as an auxiliary storage layer that extends the cloud’s computing CPU fabric. At the core of the REDS architecture is a computation-reuse
Figure 5.8: Impact of Domain Volatility and Commonality for $\alpha = 1.6$. New Epochs Signify Computation Updates. Computations Cannot Be Reused In Different Epochs.

caching scheme that stores a function’s outputs, given its inputs, i.e., records relations of: \( \text{out}_1, \ldots, \text{out}_N = \text{func}(\text{in}_1, \ldots, \text{in}_M) \). We present the elements that comprise the REDS architecture and its key mechanisms for hardware and software support.

### 5.4.1 Software Primitives and API

To preserve computation semantics, REDS must assure that the reused computation (i.e., a computation fetched from the reuse storage) would match the non-reused computation (i.e., computed via the processor), a property known as “referential transparency”. This requires expressing the input as a function of the explicit input (passed on to the called function), as well as the internal state that affects the computed output. REDS defines the following software primitives:
• inputs=<in1,in2,..,inM> The computed inputs.

• f_id The targeted function’s identifier. It is not necessary if there is only one type of computation that uses the memoization hierarchy.

• gen (optional) declares the key generation. A generation can express: (i) a logic change in the computed function (specifically, one that renders previous computation results obsolete). (ii) An internal state that might be too big to be efficiently read or too complex for the programmer to succinctly define. For example, the results of a search engine might depend on a large graph, but only a small fraction of that graph affects the search results. For such cases, generation updates serve as a time window. Previous results for the same function can still be reused until updating to a new generation.

The REDS API consists of the following architectural flows that can be integrated with a function-centric (e.g., FaaS) programming environment, or software cloud memoization environments (e.g., [35,73,144].)

• ComputeAndStore(f_id,inputs,gen): Looks up the function result in the memoization hierarchy, computes the functions in case of a miss (i.e., new computation), and stores the results in the memoization table.

• Compute(f_id,inputs,gen): Operates similar to ComputeAndStore, but does not store the outcomes in the memoization table or updates keys. This is useful for the following cases. (i) The computation is not likely to repeat (can be detected by the programmer or compiler). (ii) The input is filtered-out by a filtering mechanism, as described in Algorithm 1. In that case, the result is still needed, but it will likely pollute the cache. (iii) The input contains protected data which, when combined with a state change of the lookup and storage hierarchies, might leak sensitive information through a timing side-channel attack, which caches are susceptible to.
Figure 5.9: Top-Level REDS Architecture.

- **Delete**\(_{(f\_id, inputs, gen)}\): Deletes a key from the hierarchy and invalidates the reuse storage.

- **Store**\(_{(f\_id, inputs, gen, outputs)}\): Migration of a REDS entry from a different machine. The combination of Store and Delete can be used for storage and replication.

### 5.4.2 Design

Figure 5.9 outlines a top-level view of the REDS architecture integrated in a cloud environment. The computed input and output are communicated over the network via the network interface card (NIC). The computation fabric consists of clusters of nodes with hierarchical switching, inspired by Microsoft’s cloud acceleration microservices architecture [37]. Each node in the computation fabric consists of a NIC, a many-accelerator SoC (the dashed accelerator signifies the lack of accommodating accelerator for the requested function, highlighting an accelerator-less case), and a reuse storage module. We added the following REDS components to a traditional datacenter server.

- An In-Network Hashing Unit (INH)
that resides in the node’s NIC. The INH hashes the requested input and generates a key and a pointer. The key serves as an approximated lookup for the previously-computed input (i.e., functions as a CRC for the entire input), and the pointer corresponds to the location of the entry in REDS memory hierarchy that contains the requested input. Reuse-Driven Specialization Storage (RDSS) modules that reside in the reuse storage module. RDSS consists of memory modules that contain entries of previous function invocations, i.e., of the form: $f_{id}, \text{input}, \text{gen} \rightarrow \text{output}$. The Reuse-Storage Controller (RSC) that resides in the reuse storage hierarchy. The RSC performs a full comparison between the input stores. If the input matches, the RSC sends the corresponding output as the computed output. If there is a mismatch, the RSC triggers the CPU to compute the result and stores it in the RDSS modules.

Prior to the execution of a flow (as defined in Section 5.4.1), a meta-data packet is is transferred over the network. The packet contains a 2-bit opcode field (0x0: Delete, 0x1: Compute, 0x2 Store, 0x3 ComputeAndStore.). The packet contains other meta data fields (e.g., target compute node if controlled by software, or other platform-specific definitions.)

As Figure 5.10a shows, the INH receives the meta and input packets from the network. As the meta packet is received, a “start” signal is sent, clearing the “key hash” and “pointer hash” registers in the register file. For each computation input packet received, the hashes are computed incrementally using the received input block and the stored register values, i.e., $key = HASH(in_{blk}, key_{r})$, $pointer = HASH(in_{blk}, pointer_{r})$. The hashing logic is implemented as a XOR-gate plane, similar to previous studies [17,63]. Finally, when the input transfer is done, the key and pointer are sent to the reuse storage. As seen in Figure 5.10b, the input block are first stored in an input store memory. Once the input transfer is done, the reuse storage receives the key and input from the INH. To simplify the key management and to decouple it from congestion-driven load balancing, we co-located the key with the RDSS (i.e., input+output) entry. In the first stage, the key is compared,
Figure 5.10: Microarchitectural Diagram of the REDS Modules.

and the full input comparison takes place only if the key matches. Co-location saves timing and energy by avoiding unnecessary inputs comparison. If matching fails, the result is computed via the CPU (and stored in case it is in the filter, as seen in Section 5.3.3).

5.4.3 REDS Flows

Figure 5.11 shows all the architectural flows, for software-controlled lookups and hardware-controlled lookups. The possible flow scenarios are as follows:

- **“Compute” and “ComputeAndStore” Flows (Figure 5.11a):** 1 the generated hash key did not match the key in the pointed entry and it does not match any of the keys in the remote cluster and the flow falls back to regular computing (and update storage and lookup in case of ComputeAndStore). 2 the key was matched locally, and the REDS flow moves on to the full input comparison. 3 There was no local match, only a remove match, therefore the node performs an RDMA read for the corresponding input. In 4 the input comparison fails, which is the same scenario as 1 , which falls back to regular computation. In 4 , the key and inputs match a previous computation, and the output can be fetched and used instead of redundantly computing it via the CPU.
Figure 5.11: Pipeline Diagrams of Hardware REDS Flows. Colors Signify Different Components.

- “Store” and “Delete” Flows (Figure 5.11b): ③ Delete flow: delete the hardware-hashed key from the hierarchy, ⑦ Store flow: store the new computation output (via RDMA fetches) and associate it with the hashed key and pointer.

5.4.4 End-to-End Flow

We demonstrate an end-to-end REDS flow using a case study outlining the channel, trade-offs, and design exploration. We explore a sparse-matrix vector multiplication kernel, with packed storage format for better compression (SPMV-Ellpack [140]). While sparse linear algebra is a stable domain, compression and packing depend on the problem’s representation. As new problems are discovered, representation might change in a way that renders existing accelerator hardware obsolete (or suboptimal).

**Accelerator Design:** As REDS aims to deliver near-accelerator rates, we wish to compare its execution with a hardware accelerator optimized for the targeted computation. For our baseline, we use Aladdin [151], a timing, area, and energy modeling tool for accelerators. Aladdin enables a fast hardware/software exploration fixed-function accelerators using various tuning parameters, e.g., loop pipelining and unrolling to accommodate
software-level parallelism and partitioning of a special-purpose scratchpad memory to support memory-level parallelism. Table 5.3 shows the sweep parameters used and Figure 5.12 shows the runtime-energy sweep of an accelerator for the SPMV-Ellpack kernel.

```c
#define SPMV_ID 0x0
#define NNZ 1666
#define N 494
#define L 10

#pragma reusable fid=SPMV_ID \n input=nzval \n input=cols \n input=vec \n output=out
void spmv(double nzval[N*L], int32_t cols[N*L],
          double vec[N], double out[N]) {
    // The compiler insert the relevant call to
    // ComputeAndStore here...
    for (i=0; i<N; i++) {
        double sum = out[i];
        for (j=0; j<L; j++) {
            sum += nzval[j+i*L]*vec[cols[j+i*L]];
        }
        out[i] = sum;
    }
}
```

Snippet 5.1: Compiler-Assisted REDS Flow for SPMV

**Programming Model:** Snippet 5.1 demonstrates a compiler-assisted flow for the SPMV-Ellpack kernel, using an API similar to the OpenMP task model API [2]. The
Table 5.3: Accelerator Design Sweep Parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Sweep Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle Time [ns]</td>
<td>2, 4, 6, 10</td>
</tr>
<tr>
<td>Loop Unrolling Factor</td>
<td>1, 2, 4, 8, 16, 32</td>
</tr>
<tr>
<td>Loop Pipelining</td>
<td>Disabled/Enabled</td>
</tr>
<tr>
<td>Scratchpad Ports</td>
<td>2, 4, 8, 16, 32, 64</td>
</tr>
</tbody>
</table>

Figure 5.12: SPMV-Ellpack Fixed-Function Accelerator Exploration. Design Sweep + Optimal Points.

The programmer inserts a pragma indicating that the function is “reusable” and defines its inputs and output. At compile-time, the compiler backend inserts a call to the `ComputeAndStore` flow.\(^4\)

**Congestion-Aware CPU/Network/Memory Codesign:** One of the main challenges of devising a cloud-scale computation-reuse architecture is the ability to handle congestion. Traffic bias forms “hot spot” nodes that serve higher computation volumes than other nodes,

\(^4\)An alternative (and less prone to bugs) approach to a presented use of hardcoded function ID, is an automated hashing of the entire function code, but that might change the ID for every code change, rendering old computations obsolete.
due to the combination of runtime hashing of computed inputs and computation popularity.

Figure 5.13a demonstrates the distributions of handled computations on a 64 node cluster, for different bias rates. As the figure shows, for high bias (i.e., $\alpha = 1.2$) three nodes serve similar content due to key popularity. While load balancing is not in the scope of this work, we study how reuse memory alleviates the stress on the compute cloud nodes. Figure 5.13b motivates the use of memory by showing the impact of arrival rates of the SPMV function. The figure analyzes the handling times of three elements in the cluster node. (i) The CPU handling time is derived by execution time of the computation on the general-purpose processor. (ii) The NET handling time corresponds to the time it takes to transfer the computed input and output data on the network link. (iii) The MEM handling time corresponds to the time of reading and matching the stored input with the requested input in the reuse storage. We use the run time of a single x86 out-of-order (OoO) core with a 32KB L1 data cache and 1MB of L2 cache for “CPU”. We use the transfer times on a 56 Gb/s FDR link for “NET” and use serial input read times from a 16GB memory DRAM bank for “MEM”. The dashed lines represent the overall handling times of the compute-reuse node under different reuse storage hit rates. As the figure shows, for hit rates of 50% and higher, the node’s handling times are shorter than the network’s curve. This suggests that when designing REDS, proper sizing of the memory hierarchy amortizes compute times in a way that eliminates the computation handling bottleneck, even for modest hit rates.

**Designing REDS:** Figure 5.14 shows the different stages in the design flow. Given the cloud function characteristics (i.e., input and output sizes) and traffic bias rate, we generate input traces and run a trace-based simulation of different topologies, i.e., (different memory hierarchy size and sharing degree combinations). Using the costs of memory accessing for different banks, network data transfers, and CPU computations, we estimate the different physical costs for different topologies (e.g., different memory banking layouts, network transfer costs, etc.) for each combination of memory size and sharing degree. Fi-
nally, we sweep across the physical costs space for each of the possible topologies (i.e., total memory+sharing configuration). We demonstrate the REDS flow by evaluating it as an extension to the CPU-compute fabric on a 4096 node cloud. We explore 100 million SPMV-Ellpack computation requests, distributed with a Zipfian parameter of $\alpha = 1.3$. Figure 5.15 shows the results of the different stages in the REDS exploration flow. As shown in Figure 5.15a, we ran trace-based simulations sweeping across 91 sharing degree+memory size combinations. For the combination of topologies and physical memory layouts, we tested different memory bank sizes ranging from 1MB to 1TB DRAM modules with 4KB-128KB in word sizes. Figures 5.15b and 5.15c show the outcomes of runtime and energy exploration sweep for each topology and layout, across 2912 explored configurations. As Figure 5.15b shows, the runtime-optimized REDS topology achieves a speedup of $19 \times$ compared to the runtime-optimized accelerator. While Figure 5.15a shows that sharing improves hit rates (and as discussed in Section 5.3.3), the impact of communication across nodes is costly (the optimum is obtained for a high but not maximal degree of sharing). As Figure 5.15c shows, compared to an energy-optimal accelerator, the use of memory reduced the energy overhead of a CPU execution from more than $200 \times$ (using modest hit rates), to a total of $7.13 \times$. Compared to the optimal speedup, the optimal energy rate was attained using a smaller memory hierarchy. This was due to the increase in leakage power for large memories that resulted in diminishing returns, i.e., the impact of increased hit rate was secondary to the increase in the static energy wasted.

### 5.5 Evaluation

We demonstrate how REDS extends a cloud-scale CPU processing fabric to speed up computation and deliver near-accelerator efficiency rates. We compare REDS with a many-accelerator SoC based cloud that uses fixed-function accelerators, i.e, compare how REDS
performs versus how highly-efficient accelerators would have performed, if they were to be built.

5.5.1 Methodology

Cloud System: We evaluated REDS on a 4096 node cloud with CPU-based compute node clusters. Table 5.4 shows the network and CPU parameters of the evaluated system. Network switch and NIC parameters were taken from contemporary datasheets of cloud-serving Infiniband products [116,117], and they are also in line with previous studies on datacenter power consumption [6,22].
Workloads: We explored the computation behavior of functions from various domains taken from the Machsuite benchmark suite [140]. Table 5.5 shows the explored functions, their descriptions, input and output sizes, the execution times for runtime-optimized accelerators versus CPUs, and runtime energies for energy-optimized accelerators versus CPUs. The runtimes for OoO x86 CPUs were attained used gem5 [26] and McPAT [104]. We used Aladdin [151] to model the runtime, power, and energy, for optimized fixed-function accelerators as a comparison point. We characterized the behavior of the runtime and energy optimized accelerators by sweeping their design spaces, as shown in Section 5.4.4. As seen in the table, for most benchmarks, CPUs are slower than the accelerators by an order-of-magnitude. The few cases in which CPUs are faster are the result of added DMA transfer times for CPU-accelerator communication in a loosely-coupled accelerator in an
Table 5.4: Compute Cluster Parameters

<table>
<thead>
<tr>
<th>Switching Parameters</th>
<th>Node NIC Parameters</th>
<th>SoC Interconnection</th>
<th>Node Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same-ToR RTT 1.7 µs</td>
<td>Bandwidth 56Gb/s</td>
<td>Power 19W</td>
<td>CPU 2GHz 8-Wide OoO x86</td>
</tr>
<tr>
<td>Different-ToR RTT 3.1 µs</td>
<td>Power 100Gb/s</td>
<td>Flit Size 64-bit</td>
<td>CPU Caching 64KB L1 D-cache, 2MB L2 cache</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>Power</td>
<td>Latency 1ns</td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td></td>
<td>Energy 30pJ</td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5: Evaluated Functions. Input and Output Sizes, Runtimes and Energies for Fixed-Function Accelerators and CPUs.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Domain</th>
<th>Data Sizes[KBs]</th>
<th>Runtimes[µs] Accelerator CPU (Ratio)</th>
<th>Runtime Energies[µJ] Accelerator CPU (Ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Breadth-First Search</td>
<td>Graph Processing</td>
<td>66</td>
<td>25.6</td>
<td>26.5 (0.5×)</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier-Transform</td>
<td>Signal Processing</td>
<td>24</td>
<td>51.1</td>
<td>99.5 (1.9×)</td>
</tr>
<tr>
<td>KMP</td>
<td>Knuth Morris-Pratt Algorithm</td>
<td>String Matching (e.g., Search Engines)</td>
<td>31</td>
<td>103.1</td>
<td>301.1 (2.9×)</td>
</tr>
<tr>
<td>KNN</td>
<td>K-Nearest Neighbors</td>
<td>Data Mining</td>
<td>38</td>
<td>64.9</td>
<td>114.9 (1.8×)</td>
</tr>
<tr>
<td>MMB</td>
<td>Matrix Multiplication/Blocked</td>
<td>Linear Algebra</td>
<td>256</td>
<td>176.2</td>
<td>635.5 (3.6×)</td>
</tr>
<tr>
<td>MMC</td>
<td>Matrix Multiplication/Cubed</td>
<td>Linear Algebra</td>
<td>256</td>
<td>148.6</td>
<td>726.5 (4.9×)</td>
</tr>
<tr>
<td>MST</td>
<td>Merge Sort</td>
<td>Algorithms (e.g., Map-Reduce)</td>
<td>8</td>
<td>57.4</td>
<td>415.4 (7.2×)</td>
</tr>
<tr>
<td>MVC</td>
<td>Matrix Vector Product / Compressed Rows</td>
<td>Sparse Linear Algebra</td>
<td>25</td>
<td>43.2</td>
<td>139.9 (3.2×)</td>
</tr>
<tr>
<td>MVE</td>
<td>Matrix Vector Product / Ellpack</td>
<td>Sparse Linear Algebra</td>
<td>57</td>
<td>99.1</td>
<td>116.4 (1.2×)</td>
</tr>
<tr>
<td>S3D</td>
<td>2D Stencil</td>
<td>Image Processing</td>
<td>40</td>
<td>92.1</td>
<td>102.9 (1.1×)</td>
</tr>
<tr>
<td>S3D</td>
<td>3D Stencil</td>
<td>Computer Vision</td>
<td>64</td>
<td>179.9</td>
<td>140.7 (0.8×)</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
<td>54</td>
<td>23.89</td>
<td>96.2</td>
</tr>
</tbody>
</table>

SoC [111,152]. The table also shows that, as expected, accelerators are two to four orders-of-magnitude more energy-efficient than CPUs.

For each function, we generated three traces with 100 million requests that follow a Zipfian distribution with bias parameters: \( \alpha = 1.1, 1.3, 1.6 \). We explored a total of 33 function and traffic bias combinations.

**REDS Design Space:** For each workload, we explored the best runtime or energy configurations in the REDS design space by sweeping across different memory and sharing topologies, and for different physical layouts. Table 5.6 shows the parameters that span the explored design space. We explored about 3,000 memory size and sharing degree combinations, and a total of about 100,000 possible memory layouts. We compared
Table 5.6: REDS Design Sweep Parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Sweep Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trace Parameters</strong></td>
<td></td>
</tr>
<tr>
<td>Sharing Degree</td>
<td>1 (Private), 4, 16, ..., 4096 (Fully-Shared)</td>
</tr>
<tr>
<td>Key Filter</td>
<td>Disabled/Enabled (128 Keys w/ Random Evict)</td>
</tr>
<tr>
<td>Inter-Cluster Fetching</td>
<td>Disabled/Enabled</td>
</tr>
<tr>
<td>Total Memory Spaces</td>
<td>16GB, 32GB, ...16TB</td>
</tr>
<tr>
<td><strong>Physical Layout Parameters</strong></td>
<td></td>
</tr>
<tr>
<td>RDSS Bank Size</td>
<td>16MB-1TB DRAM</td>
</tr>
<tr>
<td>RDSS Bank Word Size</td>
<td>4KB-128KB</td>
</tr>
</tbody>
</table>

the runtime-optimized (runtime-OPT) and energy-optimized (energy-OPT) REDS with the runtime-optimized and energy-optimized accelerator.

### 5.5.2 Results

We show the results for runtime and energy for all workloads as well as the arithmetic mean (AMEAN), weighted arithmetic mean (WMEAN), and geometric mean (GMEAN) for each of the explored bias rates. Figure 5.16 shows the obtained speedups for runtime-OPT REDS, and Figure 5.17 shows the relative energies for energy-OPT REDS. For the optimized REDS configurations, we differentiate between the costs of memory handling, CPU execution, and network traffic and congestion. Figure 5.18 shows the breakdowns of running times for runtime-OPT REDS, and Figure 5.19 shows the breakdowns of the energy spent in an energy-OPT REDS.

An important aspect to account for in the design exploration, is that the accelerator-based clouds have lower networking costs. In contrast to REDS, accelerator execution is stateless, since there are no “hot spots” formed by bias-induced congestion. As we show in Section 5.4.4, by leveraging low memory costs, REDS amortizes overall computation costs, while accounting for bias-induced congestion of popular keys.
**Results Summary:** Compared to the runtime-optimized accelerators, runtime-OPT REDS speeds up execution by an average of 2.9× for low bias workloads ($\alpha = 1.3$), and an average of 20× for high bias workloads ($\alpha = 1.6$). Compared to the energy-optimized accelerators, energy-OPT REDS achieves an average energy overhead of 48× for low bias, and an average of 3× for high bias. In some high bias cases, REDS even achieves lower runtime energy than the accelerators. These were the cases that the relative runtime energy of the CPU was relatively low at just two orders-of-magnitude higher than the fixed-function accelerator (e.g., 103× for KNN).

**Runtime and Energy Bottlenecks:** As Figure 5.18 shows, for low bias, the majority of the running time (86%) is spent in the CPU. As bias increases the majority of running time shifts towards the memory, which accounts for an average of 85% of overall execution for the high bias workloads ($\alpha = 1.6$). At this point, the hit rate and costs of memory lookup and fetching increase, i.e., $P_{HIT} \uparrow, C_{LOOKUP} \uparrow, C_{FETCH} \uparrow$. As a result, overall memory costs
are almost the same as the amortized timing cost of the CPU computation, i.e., \( P_{\text{HIT}} \times (C_{\text{LOOKUP}} + C_{\text{FETCH}}) \approx (1 - P_{\text{HIT}}) \times C_{\text{FUNCTION}} \). Therefore, an increase in hit rate would result in diminishing returns. As Figure 5.19 shows, from an energetic perspective, the costly CPU accounts for 97\% of overall runtime energy in low bias traffic, but as bias increases the majority of energy is spent in the network (74\% for the high bias traffic), situating the network as the energetic bottleneck for high bias traffic.

**Forward-Looking on Reuse-Driven Specialization for Accelerator-less Clouds:**
The slowdown of CMOS scaling makes general-purpose processors costly [56], and situates deployment of accelerators as one of the prime means to improve computation
efficiency [170,178]. Efficiency is particularly important in cloud environments that cater to continuously-rising computational demands on a planetary scale [113]. However, the rapid changes and heterogeneity of cloud workloads [93] makes accelerators untenable for emerging applications or workload changes. As our experiments show, by leveraging the characteristics of cloud-serving traffic, the reuse-driven specialization flow uses programmable memory and CPUs to speed up computations. REDS can therefore achieve near-accelerator energy efficiency. Compared to fixed-function accelerators, REDS reduces the average energy overhead of inefficient (yet applicable) CPUs from 230× in Table 5.5 to only 3× for biased traffic, extending the CPU fabric without being dependent on accelerator deployment times. The emergence of paradigms such as serverless FaaS computing would make cloud workloads more volatile and more function-centric, paving the way for reuse-driven specialization for future clouds.

5.6 REDS Conclusions

While accelerators deliver high compute efficiency rates in modern cloud architectures, their use is not always feasible. Emerging domains are left accelerator-less in the time between the identification of computation needs and the deployment of the first accelerators. Volatile domains can remain accelerator-less due to rapidly-changing functionality that is unsupported by existing accelerator hardware. We present REDS, an architecture that employs reuse-driven specialization for accelerator-less clouds. REDS leverages the skewed nature of cloud traffic to deploy computation-reuse memory layers to memoize the results of computed functions in a versatile and microarchitecturally-agnostic manner. We discuss the implications of accelerator-less workloads in modern cloud environments, and the challenges for cloud-scale computation-reuse. We showed that when combined with general-purpose processors, reuse-driven specialization can both speed up computations and improve efficiency to achieve near-accelerator efficiency rates without compromising
flexibility or being dependent on accelerator deployment times. As cloud programming environments become more function-centric and push towards rapidly-changing cloud functionality, we believe that reuse-driven specialization will become a necessity.
Chapter 6

Conclusions

This dissertation has presented and tackled the limitations of accelerator-centric architectures.

We showed that while accelerators were motivated by transistor-imposed limitations, they are empirically and inherently dependent on the returns of improving transistor technology. Since specializing accelerator chips is conceptually limited, once transistors stop improving accelerator designs will become fixed-budget optimizations, and converge to near-optimum until ultimately delivering diminishing returns when reaching an *accelerator wall*.

To overcome the accelerator wall, we explored still-scaling emerging memory technologies and constructed COREx, a compute-reuse architecture for accelerators that extends chip-based specialization not only technologically by using “beyond-CMOS” memory cells, but also conceptually, by specializing memory layers to the accelerator and to the application. We showed how contemporary models of STT-RAMs, ReRAMs, PCMs and Racetrack memories can already show promising improvements, and predict that as memory technology improves, memoization-centric specialization will become widespread.
Finally, we leveraged the architecturally-agnostic nature of memoization to devise a flexible reuse-driven specialization architecture that is integrated with programmable cores and can achieve comparable returns to highly-efficient non-programmable accelerators.

6.1 Future Work

Going forward, we see several outcomes of the conducted studies that will impact future specialized architectures.

6.1.1 Memoization Flows and Tools

Memoization programming primitives and flows will be integrated with future architectures. The motivation for incorporating memoization with contemporary accelerators is twofold. First, as the evaluation of this dissertation shows, memoization for accelerators is beneficial even when using existing non-volatile memory models. Second, from an engineering perspective, if an application was important enough to have a custom-built accelerator programmed to it, it is worthwhile to explore every optimization that improves its timing or energy efficiency, including memoization and memory-layer specialization.

We project the resurfacing of memoization frameworks and tools following the rise of function-centric clouds that rely on serverless architectures and function-as-a-service (FaaS) applications. We strive to develop tools and flows that use contemporary memory and storage models for functional-centric clouds while leveraging the ubiquitous computation redundancy in datacenters.

6.1.2 Scalable Architectures with Emerging Non-Volatile Memories

Scalable technologies such as emerging non-volatile memories are becoming increasingly prominent in modern architectures, and in accelerators in specific. As current studies show, non-volatile memories show promise for future scaling. Following the end of Moore’s
Law, non-volatile memory technologies are expected to gain more traction, and once new
memory roadmaps are established, they will change the way accelerators are built.

We envision a co-design of NVM-based memoization and accelerators that strives to
maximize not only traditional metrics, such as performance or energy efficiency but also the
future scalability of outcoming designs while using the principles and flows we presented.
Bibliography


[21] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang,


[82] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. Programming and runtime support to blaze FPGA accelerator deployment at datacenter scale. In ACM Symposium on Cloud Computing (SoCC), SoCC ’16, pages 456–469, New York, NY, USA, 2016. ACM.


[89] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-Luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu,


[117] Mellanox. Sb7800 infiniband edr 100gb/s switch system.


