Managing Data Movement Bottlenecks in Accelerator-Rich Systems

by

Sudhanshu Gupta

Submitted in Partial Fulfillment of the

Requirements for the Degree

Doctor of Philosophy

Supervised by Professor Sandhya Dwarkadas

Department of Computer Science

Arts, Sciences and Engineering

Edmund A. Hajim School of Engineering and Applied Sciences

University of Rochester

Rochester, New York

2025

To my parents and my sister

Table of Contents

Biographical Sketch vii

Acknowledgments ix

Abstract xii

Contributors and Funding Sources xiii

List of Tables xiv

List of Figures xvi

1 Introduction 1

1.1 Motivation 2

1.2 Research Contributions 5

1.3 Dissertation Organization 7

2 Background 9

2.1 Domain Specific Accelerators 10

2.2 Processing in Memory 17

2.3 Emerging Memory Technologies 21

2.4 GPU/Host Communication 23

2.5 Large Language Models 24

2.6 PIM on Emerging Memory Technologies 27

3 Relieving Memory Pressure In SoCs Via Data Movement-Aware Accelerator Scheduling 29

3.1 Background 32

3.2 RELIEF: Relaxing Least-laxity to Enable Forwarding 39

3.3 Evaluation Methodology 49

3.4 Results 52

3.5 Related Work 67

3.6 Summary and Discussion 69

4 Concurrent PIM and Load/Store Servicing in PIM-Enabled Memory 71

4.1 Background 74

4.2 Evaluation Methodology 77

4.3 Characterizing GPU/PIM Interference 83

4.4 Memory Access: Interconnect Bottlenecks 85

4.5 Memory Utilization: Scheduling Bottlenecks 89

4.6 First Mode-FR-FCFS (F3FS) - New and Improved PIM-Aware Memory Access Scheduling 95

4.7 Related Work 101

4.8 Summary 103

5 On The Impact of Emerging Heterogeneous Memory on Accelerator Performance 104

5.1 Background 106

5.2 Evaluation Methodology . . . 109
5.3 GPU/Host Data Movement Characterization . . . 111
5.4 Impact of Weight Placement . . . 117
5.5 Related Work . . . 128
5.6 Conclusion . . . 133

6 Conclusion 135

6.1 Future Work . . . 137

Bibliography 139

A Chapter 3 Artifact Appendix 194

A.1 Abstract . . . 194
A.2 Artifact check-list (meta-information) . . . 194
A.3 Description . . . 195
A.4 Installation . . . 196
A.5 Experiment workflow . . . 197
A.6 Evaluation and expected results . . . 197

B Chapter 4 Artifact Appendix 198

B.1 Abstract . . . 198
B.2 Artifact check-list (meta-information) . . . 198
B.3 Description . . . 199
B.4 Installation . . . 200
B.5 Experiment workflow . . . 201
B.6 Evaluation and expected results . . . 201

C Chapter 5 Artifact Appendix 203

C.1 Abstract 203

C.2 Artifact check-list (meta-information) 203

C.3 Installation 205

C.4 Evaluation and expected results 205

Biographical Sketch

Sudhanshu Gupta was born in New Delhi, India. He graduated with a Bachelor of Technology in Computer Science and Engineering from NIIT University, India in 2019. His first foray into computer architecture was when he worked with Professor Sanjay Gupta on the design and implementation of multi-cycle CPU on an FPGA. In 2018 and 2019, he interned at the Indian Institutes of Technology in Kanpur and Delhi, working on performance characterization of machine learning and vision workloads on CPUs and GPUs. These experiences were a crucial motivator for him to pursue graduate studies in computer architecture. He started his graduate studies at the University of Rochester in 2019, conducting research on accelerator-rich systems with Professor Sandhya Dwarkadas and earning a Master of Science in Computer Science in 2021. In 2022, he interned at AMD Research, looking at the design of processing in memory enabled systems.

The following publications were a result of his work during the doctoral study:

1. S. Gupta and S. Dwarkadas, “RELIEF: Relieving Memory Pressure In SoCs Via Data Movement-Aware Accelerator Scheduling,” *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Edinburgh, United Kingdom, 2024, pp. 1063-1079, doi: 10.1109/HPCA57654.2024.00084.

2. S. Gupta, N. Madan, S. Puthoor, N. Jayasena, and S. Dwarkadas, “Concurrent PIM and Load/Store Servicing in PIM-Enabled Memory,” *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Ghent, Belgium, 2025, pp. 320-334, doi: 10.1109/ISPASS64960.2025.00037.

3. S. Gupta and S. Dwarkadas, “Improving the Performance of Out-of-Core LLM Inference Using Heterogeneous Host Memory,” *2025 IEEE International Symposium on Workload Characterization (IISWC)*, Irvine, CA, USA, 2025.

Acknowledgments

I would like to thank everyone who supported me not only during graduate school, but also people who got me here in the first place.

My family showed incredible courage and support when I accepted University of Rochester's offer for graduate school. Being separated for prolonged periods, especially during the pandemic, posed a significant emotional challenge. However, they stood by me and offered unwavering support. I thank my mother for teaching me compassion, my father for teaching me hard work, and my sister for teaching me resilience.

None of the work in this dissertation would be possible without my advisor, Professor Sandhya Dwarkadas. Sandhya gave me the freedom to explore different research areas before narrowing down on one for my dissertation. She patiently worked with me on paper submissions and made sure I received all the time and feedback I needed. Throughout graduate school, she encouraged me to think both deep and broad: understand every little detail of what I am working on while also keeping an eye out on what is happening beyond. The rigor of this education has changed me for the better.

I cannot imagine a department more student friendly and supportive as the Department of Computer Science. Professor Michael Scott, the former chair of our department and a member of my committee, has fostered a student-first environment that is always rooting for the success of its students above anything else. Michael has been a great source of inspiration, both professionally and personally. I thank the department staff, both past and present, including Abigail Burton, Amanda Rigolo, Dave Costello, Eileen

Pullara, Emily Tevens, Ian Ward, Jim Roche, Kristi Kongmany, Mary Méndez-Rizzo, Michelle Kiso, Nick Quattrociocchi, Robin Clark, and Shelley Zoeke for working tirelessly to help students navigate through all the administrative requirements. I would also like to thank my committee members Professor Yuhao Zhu and Professor Kevin Skadron. Yuhao, who was also my first year advisor, helped me hone my presentation skills and taught me to think broadly about the problems I solve, even if it goes beyond my area of expertise. Kevin taught me essential writing skills and how to frame my research better.

Balancing work and life can be difficult in graduate school. I would not know because my partner, Rashi, made it so easy for me. She inspired me to turn my five minute breaks into hour long ones to help me focus better. She is kind, empathetic, and went to great lengths to help me succeed in graduate school. I thank her for soothing and supporting me through tough times and celebrating my successes more than I would. She continues to challenge my world views and is a big part of who I am today. Our cat, Idli, is the softest and sweetest companion I could ask for. I thank her for accompanying me during late night writing sessions.

Graduate school can be lonely and difficult. I have been blessed with some incredible friends who were there with me through thick and thin: Dhanush Bhatt, Mingzhe Du, Xiaofei Zhou, Vasisht Guru Prasad, Abhishek Tyagi, Adiba Proma, Benjamin Valpey, Komail Dharsee, Arvind Srivatsava, Meghana Murthy, and Purvanshi Mehta. Dhanush and Mingzhe have been my constants throughout graduate school. I enjoyed gaming nights with Dhanush and weekly lunches with Mingzhe. They both stuck with me through some very tough times: PhD milestones, a pandemic, heartbreaks, and mental health struggles. Their companionship has been instrumental in surviving graduate school. Xiaofei is easily the most hard working and inquisitive person I have ever met. I thank her for our deep conversations and her thoughtful and beautiful gifts. I adore Vasisht's ability to be high energy in parties and be calm during serious conversations. I enjoyed his surprise evening visits to my place. Abhishek reminded me of home far from home. I thank him for taking me to new places and for finding the best chole bhature. I will miss his physical humor. Adiba is a very sweet person who is always looking out for her friends. I love her dark and dry humor and thank her for slapping Abhishek for his ill-timed humor. Benjamin was a great office mate and I enjoyed my time taking courses and working through assignments with him. I thank him for our board game nights with Natalie and Powderpuff, and his very passionate rants about the state of the world. Komail was the one person who most people in the department knew of when I first came, and for good reasons. He was always present for anyone who needed help and contributed in more ways to the department than I can list here. I thank him for cold and rainy day BBQs (however exhausting they were), foosball matches, gaming nights, dating conversations, and Costco visits. I thank Arvind for making the pandemic less lonely and for learning all the board games we enjoy. I thank Meghana and Purvanshi for the late night hangouts and dinners, and for exploring new things with me in the city.

Abstract

Data movement overheads are a key performance bottleneck that limit the efficiency of modern computer systems. Domain-specific accelerators help ameliorate some of these bottlenecks by improving data reuse and overlapping computation with communication. Yet, the resource intensive nature of contemporary workloads like computational photography and machine learning applications continues to strain the surrounding memory subsystem, limiting scalability and performance.

This dissertation identifies three fundamental communication bottlenecks: inter-accelerator data movement, accelerator/memory data movement, and limited memory capacity. Traditional inter-accelerator communication via the main memory impedes system scaling as memory access time dominates execution time. Concurrently, growing application working set sizes face both performance and capacity limitations at the host memory. This dissertation tackles these three issues separately. First, a data movement-aware accelerator scheduling policy is proposed that maximizes the utilization of accelerator-to-accelerator communication hardware, reducing main memory and interconnect pressure. Second, architectural innovations are presented to enhance the viability of processing in memory architectures that eliminate such communication entirely by moving compute closer to memory. Finally, a study characterizing the implications of heterogeneous host memory on accelerator performance is presented, serving as the basis for smarter data placement schemes for outsized workloads like large language models. When combined, the contributions of this dissertation enhance the efficiency and scalability of future accelerator-rich systems.

Contributors and Funding Sources

This work was supervised by a dissertation committee consisting of Professors Sandhya Dwarkadas, Michael Scott, Kevin Skadron, and Yuhao Zhu.

The research presented in Chapters 3 and 5 is based on work previously published by Sudhanshu Gupta and Sandhya Dwarkadas [95, 96]. The work in Chapter 4 is based on work previously published by Sudhanshu Gupta, Niti Madan, Sooraj Puthoor, Nuwan Jayasena, and Sandhya Dwarkadas [97].

This material is based upon work supported in part by the National Science Foundation Award CNS-1900803 and the University of Rochester. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of above named organization.

List of Tables

3.1 Elementary accelerators used in this work. SPAD stands for local scratchpad memory. 35

3.2 Absolute time spent in compute vs data movement. These are sum totals and do not account for computation/communication overlap. . . . 36

3.3 DAG node data structure 47

3.4 Accelerator metadata 48

3.5 Vision and machine learning benchmarks 49

3.6 Simulation setup 50

3.7 Number of finished DAGs in each application mix under continuous contention. 63

3.8 Accuracy of compute time and data movement predictors, along with the accuracy and performance of memory bandwidth predictors. Negative error values represent underestimation of true value while positive error values represent overestimation. The geometric mean uses absolute error values. 64

4.1 Simulation parameters 78

4.2 PIM benchmarks 79

4.3 GPU benchmarks 80

5.1 System configuration 110

5.2 LLM model/memory configuration 111

5.3 CXL configurations 126

5.4 Overlap of compute and communication with different weight allocation policies under NVDRAM configuration and the three different CXL configurations. A ratio of 1 indicates perfect overlap, while lower and higher values indicate memory-boundedness and compute-boundedness, respectively. 127

List of Figures

1.1 Overview of the contributions of this dissertation. 6

2.1 Different types of accelerator coupling. 11

2.2 Performance/overhead spectrum of processing in memory. 17

2.3 High level architecture of bank-level PIM prototypes built by Samsung (a) and SK Hynix (b). RF: register file, MAC: multiply and accumulate, AF: activation function. 18

2.4 CXL Type 3 devices enable transparent, coherent, and technology agnostic expansion of main memory. 23

2.5 Architecture of a decoder-only LLM. Prefill and decode perform GEMM and GEMV operations, rendering them compute and memory bound, respectively. 25

3.1 Kernels in different image processing and RNN applications. 33

3.2 Comparison of FCFS, GEDF-D, GEDF-N, LAX, and HetSched to an ideal schedule. RELIEF achieves the ideal schedule. Brown and green arrows represent forwarding and colocation, respectively. The "Accelerators" box indicates the available number of each accelerator type. . . 40

3.3 System architecture depicting the hardware manager and the interconnect. 46

3.4 Percent of total forwards and colocations, computed as the ratio of the total number of forwards/colocations to the total number of edges in the mix. 53

3.5 Breakdown of data movement into main memory traffic (lower bars), SPAD-to-SPAD traffic (upper bars), and colocations (empty space). Data is normalized to total data movement when all loads and stores go to main memory. 55

3.6 Total main memory and scratchpad memories' energy consumption under high contention using gem5-SALAM's energy models. 56

3.7 Accelerator occupancy is defined as ratio of the sum of total of all accelerators' compute time to the the end-to-end system execution time, measured from the initiation of all applications to the completion of the last application. Higher is better. 57

3.8 Percent of node deadlines met. 59

3.9 Slowdown (a) and DAG deadlines met (b) under high contention. 61

3.10 Slowdown (a) and DAG deadlines met (b) under continuous contention. 62

3.11 Impact of memory predictors on missed deadlines under high contention. 65

3.12 Average (bars) and tail (lines) latency of the scheduler with different policies on a Cortex-A7 based microcontroller, under high contention. . 65

3.13 RELIEF's sensitivity to system interconnect under high contention. Interconnect occupancy is defined as the percentage of cycles for which the interconnect had at least one transaction going through. 66

4.1 PIM-enabled GPU architecture. Each HBM layer contains eight functional units (FUs) that are shared by two banks each (fewer shown in figure for readability). 72

4.2 PIM functional unit (FU) microarchitecture. The SIMD ALU can implement generic math and logic operations and/or domain specific operations. 75

4.3 Vector addition PIM kernel. PIM kernels have a block structure, where a block consists of consecutive PIM operations to the same row. The size of the block is usually a multiple of the register file (RF) size (*n*). . 76

4.4 Memory access characteristics of the Rodinia benchmark suite, running on 80 and 8 SMs, and the PIM kernels in terms of (a) interconnect request arrival rate, (b) DRAM request arrival rate, (c) DRAM bank-level parallelism (BLP), and (d) DRAM row buffer hit rate (RBHR). The high whiskers are labeled with the most intensive kernel for that metric. 84

4.5 Average speedup of Rodinia benchmark suite when running on 72 SMs and when co-executing with four memory intensive kernels. 85

4.6 Comparison of the baseline memory subsystem (a) with our proposed changes (b). 86

4.7 MEM request arrival rate into the memory controller, without (a) and with (b) separate MEM and PIM virtual channels, normalized to standalone execution (higher is better). 87

4.8 Switching between MEM and PIM modes leads to loss in locality since the two request types often map to different rows, as seen for requests mapping to Rows X (PIM) and B (MEM). MEM→PIM switches also suffer from bank idle time, like Bank 1 in the figure, since MEM requests on different banks execute asynchronously. 89

4.9 Fairness (a) and throughput (b). 91

4.10 Average number of mode switches (a) and MEM→PIM switch overheads in terms of additional MEM conflicts (b) and the latency of draining the MEM queue (c). 92

4.11 LLM speedup for each policy with both a combined and separate VCs. The speedup is normalized to sequential execution of QKV generation and multi-head attention, while the *Ideal* represents the minimum of the two stages. 94

4.12 Hardware overheads of F3FS in terms of the mode switch logic complexity, compared to FR-FCFS. 96

4.13 Fairness (a) and throughput (b) of a compute intensive (G10) and four memory intensive (G6, G11, G17, G19) Rodinia kernels, averaged across all PIM kernels. 98

4.14 (a) Impact of F3FS components on fairness index (FI) and system throughput (ST) of P2 and speedup of the LLM. The shaded and non-shaded ST regions represent MEM and PIM speedups, respectively. (b) Sensitivity of F3FS to interconnect queue size under VC2 configuration in terms of FI and ST across all GPU/PIM combinations. 100

5.1 Host/GPU memory copy bandwidth. The numbers 0 and 1 represent the two NUMA nodes. 112

5.2 Time to first token (TTFT), time between tokens (TBT), and throughput (tokens/s). 113

5.3 Compute/communication overlap during prefill and decode stages. The bars represent average weight transfer time while the line represents average compute time. The horizontal dashed line represents the ideal weight transfer time on an all-DRAM system. Note the different scales for the two y-axes in (b) and (d). 115

5.4 Compute/communication overlap during prefill and decode stages with compression. The bars represent average weight transfer time while the line represents average compute time. The symbol *(c)* represents compressed configurations. 116

5.5 Per-layer weight load latency for a subset of OPT-175B layers (70/194) (a) and weight distribution of multi-head attention (MHA) and feed forward network (FFN) layers in SSD/FSDAX (b) and NVDRAM/MemoryMode (c) configurations. 119

5.6 Overlap of MHA/FFN compute with the transfer of FFN/MHA weights in the prefill stage of OPT-175B with compression enabled. The bars represent average weight transfer time while the line represents average compute time. The overlap in decode stage with both batch sizes is nearly identical to prefill with batch size 1. 121

5.7 (a) Breakdown of HeLM's weight distribution across host and GPU. The number under each weight is the uncompressed/compressed size of the weight. (b) HeLM's weight distribution. 121

5.8 Impact of HeLM on (a) compute/communication overlap during decode and (b) time to first token (TTFT) and time between tokens (TBT). This is evaluated using OPT-175B with a batch size of 1. In Figure (a), the bars represent average weight transfer time while the line represents average compute time. 123

5.9 Performance impact of All-CPU weight allocation on OPT-175B. Figures (d) and (e) compare compute/communication overlap with baseline weight allocation and batch size 8 to All-CPU and batch size 44. In both the figures, the bars represent average weight transfer time while the line represents average compute time. 124

5.10 Projected performance improvements offered by HeLM (batch size=1) (a) and All-CPU (b) on CXL-based systems using OPT-175B. 128

1 Introduction

The growth and ubiquity of computing in modern life has had an undeniably positive impact on our society, both economic [208] and social [82]. For a long time, technology scaling ensured that the compute capability of modern systems nearly doubled every year [191], underpinning the development of applications like large language models (LLMs) [64, 113]. Even as technology scaling hits physical limits [151], however, demand for computing power shows no signs of slowing down [267].

In order to keep up with increasing performance and energy efficiency demands, modern systems have come to rely increasingly on specialized hardware accelerators [266]. This trend is stimulated by the end of Dennard scaling [65], which stipulated that shrinking transistor sizes result in a proportionate reduction in power consumption, and the consequent increase in the amount of “dark silicon” [71], i.e., transistors that need to be turned off due to insufficient power. Accelerators trade CPUs’ general purpose processing capabilities for a leaner, application-specific architecture that is tailored to maximize data reuse and is rid of instruction overheads like fetch and decode [102, 220] This allows them to achieve orders of magnitude improvement in performance while consuming a fraction of power of a traditional high-performance CPU [2, 46, 55, 57, 150, 245, 264].

Accelerators optimize for operational intensity (i.e., operations per byte of data) by fusing multiple operators and maximizing data reuse. However, data movement between accelerators and between accelerators and memory continues to be a challenge. These problems are exacerbated by two application trends: 1) rising computational demands that make accelerator chaining lucrative [57, 67, 90], resulting in increased inter-accelerator communication, and 2) an explosion in application working set sizes that stress both memory performance and capacity [169, 283]. This dissertation tackles each of these issues via a combination of hardware and software optimizations, with the simultaneous goals of improving system efficiency and minimizing implementation costs of potential solutions.

1.1 Motivation

Accelerators predominantly sit outside CPUs and operate asynchronously in a loosely-coupled fashion. This allows accelerators to be designed in isolation from CPUs and marketed as intellectual property (IP) cores. Systems-on-chip (SoCs) designers typically assemble multiple IPs, from potentially different manufacturers, to compose complete SoCs [242]. In order to maximize performance and accelerator-level parallelism [228], applications often chain accelerators in a producer/consumer fashion, effectively pipelining the computation across multiple accelerators [196].

Traditionally, data communication between pairs of accelerators is performed via the main memory [304]. While tenable for few accelerators and small data sizes, the contention for on-chip network and main memory bandwidth through such communication presents a serious performance bottleneck as systems become more heterogeneous [57] and data sizes grow [278]. Contemporary techniques to reduce this contention include ARM AXI-Stream [17, 24], which allows multiple producer/consumer buffers to be connected over a crossbar switch, and Linux P2PDMA [172, 240], which enables direct DMA transfers between PCIe devices. These solutions enable *forwarding* of data between accelerators, keeping data on-chip and reducing contention for main memory. Though data forwarding can lower memory and interconnect congestion for a given application, it can negatively influence the quality-of-service (QoS) for other competing applications by making accelerators less available.

Challenge #1: The efficient use of inter-accelerator data movement optimizations while providing quality-of-service (QoS) to competing applications remains an open challenge.

As data sizes grow, application performance across usage domains are becoming increasingly memory bound, ranging from consumer mobile applications [33] to large server workloads [75, 276]. This is a result of not only increasing working set sizes and advent of data hungry applications like large language models, but also of technological challenges in DRAM scaling [192]. In particular, processor performance has always scaled faster than memory performance leading to the latter becoming the predominant performance bottleneck and giving rise to the problem of "memory wall" [110, 287]. While newer memory technologies like HBM [131, 139], GDDR6 [132], and HMC [100, 130] reduce the memory bottleneck by offering wider links and increased parallelism and scalability [39], they still struggle to close the gap between processor and memory performance. Domain-specific accelerators also ameliorate this problem to a certain extent by optimizing for on-chip data reuse and overlapping computation with communication [217], but the widening processor/memory performance gap necessitates a more radical rethinking of how we see computing.

Processing in memory (PIM) [3, 84, 99, 108, 161, 195, 270] has emerged as a promising solution to the problem, advocating for compute to be moved closer to data instead of the other way around. PIM architectures place compute units close to/inside main memory cells, minimizing data movement costs and achieving wide data parallelism. While PIM-enabled memories offer significant performance and energy improvements over conventional architectures, integration of such memories into existing systems remains an open challenge. In particular, naively retrofitting PIM-enabled memory in a conventional processor could be detrimental to its performance (Section 4.3). PIM applications are optimized to saturate the memory subsystem to maximize speedup. Since modern systems are often multi-programmed, such saturation can lead to extreme unfairness and denial of service to other non-PIM applications.

Challenge #2: Incorporating PIM-enabled memories into existing memory hierarchies needs careful re-design and management of shared resources to maximize memory throughput and ensure fairness.

Memory density is another key constraint of contemporary DRAM technology. Applications like state-of-the-art LLMs comprise of trillions of parameters that are needed at runtime [190], requiring terabytes of memory. In addition to the model parameters, LLMs also maintain a key-value store of past queries and transient layer activations. The latter two grow dynamically as the model performs inference on incoming queries, constituting as much as 35% of the total memory requirement for running inference [154]. The rising capacity demands of such applications has far surpassed the rate at which DRAM density has grown [87, 169]. Emerging memory technologies like phase change memory (PCM) [221], resistive RAM (ReRAM) [45], and spin-transfer torque RAM (STT-RAM) [19, 158, 223, 255] improve density compared to traditional DRAM while achieving varying degrees of performance parity. PCM, in particular, has been commercialized as Intel Optane DCPMM [50, 121], a byte-addressable persistent memory module that fits into regular DDR4 slots (albeit using a custom protocol called DDR-T [122, 307]) and provides regular load/store semantics. While semantically consistent, Optane has different architectural characteristics: 3x and 4x lower read and write bandwidth, non-linear bandwidth scaling, and internal access granularity of 256 bytes [129, 218]. Besides memory technology innovations, interconnect technologies like compute express link (CXL) [60] allow for coherent and technology-agnostic expansion of main memory over PCI Express, creating large memory pools that can be shared by multiple processors and devices. However, even with optimizations to the link layer, CXL adds at least 70 nanoseconds to round-trip memory access latency [252].

The performance characteristics of such emerging technologies have led to the development of data layout optimizations to address both performance and capacity limitations [4, 167, 186, 224, 273, 283]. Accelerators interface with host memory to not only load initial data, but often also spill data when they run out of local memory, especially for large footprint applications like LLMs [86, 254, 292]. In such instances, data placement across the memory hierarchy can be a key factor in data movement costs and can affect the balance of computation time versus communication time.

Challenge #3: Effective use of emerging host memory technologies by accelerators requires careful application-specific data placement to adequately overlap data movement costs with computation.

Overcoming the three challenges discussed above are critical to the design of efficient accelerator-rich systems. This dissertation attempts to address each of them via a variety of hardware and software solutions.

1.2 Research Contributions

Thesis statement: Data movement remains a critical performance bottleneck in modern accelerator-rich systems. In particular, inter-accelerator and accelerator/memory data movement are key impediments to application and system scalability. This thesis demonstrates that: 1) communication-aware accelerator scheduling can reduce inter-accelerator data movement overheads while maintaining quality of service; 2) smarter memory access scheduling in PIM-enabled systems can enhance system-level fairness and throughput; and 3) latency tolerance-aware data placement can effectively overlap the cost of data movement from high capacity heterogeneous host memory with computation. Put together, these solutions aim to improve the efficiency and scalability of accelerator-rich systems.

Figure 1.1 presents an overview of this dissertation's primary contributions, which are elaborated below.

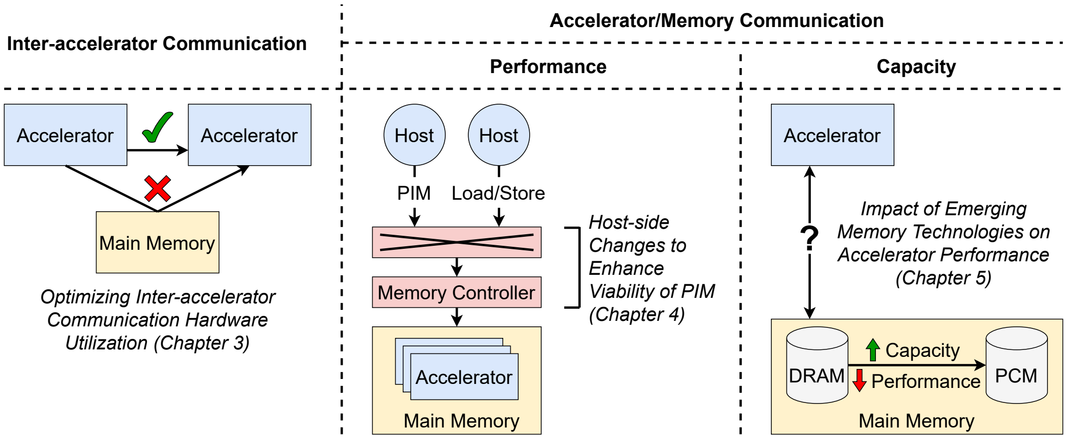

Inter-accelerator Communication
Accelerator/Memory Communication
Performance
Capacity
Accelerator
Accelerator
Main Memory
Optimizing Inter-accelerator Communication Hardware Utilization (Chapter 3)
Host
Host
PIM
Load/Store
Memory Controller
Accelerator
Main Memory
Host-side Changes to Enhance Viability of PIM (Chapter 4)
Accelerator
?
Impact of Emerging Memory Technologies on Accelerator Performance (Chapter 5)
DRAM
Capacity
Performance
PCM
Main Memory

Figure 1.1: Overview of the contributions of this dissertation.

- **Optimizing inter-accelerator communication:** Modern SoCs often incorporate inter-accelerator communication protocols, but their utilization is left to programmers. This dissertation proposes RELIEF, an intelligent accelerator scheduling policy that considers data movement tax as a first-class design goal. RELIEF schedules accelerator requests from competing real-time applications in a way that maximizes *forwarding* of data from producer accelerators to consumer accelerators using on-chip communication protocols without going through the main memory. The key idea is to exploit the slack time (time margin to deadline) of one application to prioritize requests from another application such that the latter's producer and consumer requests can be scheduled in consecutive order, avoiding the need to temporarily write producer results to main memory. RELIEF is an online policy that keeps a live record of each application's slack time, minimizing missed deadlines due to such forwarding.

- **Enhancing viability of PIM:** Processing in memory (PIM) effectively eliminates data movement between compute and memory by moving compute near/inside memory cells. This dissertation presents the observation that retrofitting PIM into existing systems can degrade throughput and cause unfairness between PIM and non-PIM applications, primarily caused by a saturation of resources by the former. Based on this observation, modifications to on-chip interconnects and a new memory controller scheduling policy, called F3FS, are proposed to remedy the problem. The interconnect modifications separate PIM and non-PIM requests into different channels, eliminating interference. At the memory controller, F3FS ensures each request type gets proportionate access to memory resources for fairness, while minimizing switching between them to maximize throughput.

- **Quantifying impact of emerging memory technologies:** High capacity but low performance memory architectures like Intel Optane DCPMM [50, 121] and CXL-based memory expansion [62] can serve as an effective spill memory for large data applications like LLMs executing on GPUs. This dissertation presents a characterization of host/device interactions in such heterogeneous architectures using a real Intel Optane based machine, highlighting the importance of careful data placement and balancing of compute and communication to hide the performance overheads of such memory. The characterization serves as the basis for two application-aware data placement schemes, one each optimizing for latency and throughput.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 provides a background in all three broad tenets of this dissertation: accelerator design and chaining, processing in memory, and emerging memory technologies. Chapter 3 presents RELIEF, an online accelerator scheduling policy that opportunistically forwards data between producer and consumer accelerators while ensuring fairness between competing applications. Chapter 4 characterizes performance bottlenecks when integrating PIM-enabled memory into existing systems. It then presents hardware optimizations at the interconnect and memory controller for efficient GPU/PIM co-processing under both competitive and collaborative scenarios. Chapter 5 compares host/GPU communication performance of heterogeneous host memory to traditional DRAM memory. This characterization serves as the basis for improved data placement schemes that retain the capacity benefits of heterogeneous memory while closing the performance gap to DRAM memory. Chapter 6 concludes the dissertation with potential future work directions.

2 Background

This chapter covers essential background to the bottlenecks described in Chapter 1 and the contributions presented in the following chapters. Section 2.1 motivates the need for domain specific accelerators, emphasizing the key features that enable them to outperform traditional CPUs and the different ways they can be integrated. Crucially, the section summarizes key hardware and software impediments to accelerator chaining and prior solutions that address them. Section 2.2 presents a classification of different processing in memory (PIM) architectures and the tradeoffs that each of them entail. This is followed by a brief discussion on the various PIM programming paradigms. Section 2.3 looks at some emerging memory technologies that offer higher capacity compared to traditional DRAM at the cost of performance. In particular, the section covers background on the architecture and performance of Intel Optane and CXL-enabled memory expansion. Section 2.4 summarizes different GPU/host communication paradigms and programming interfaces. Section 2.5 discusses the architecture of modern large language models (LLMs), highlighting the challenging nature of LLM inference. Finally, Section 2.6 provides an overview of recent proposals for incorporating PIM into emerging memory technologies.

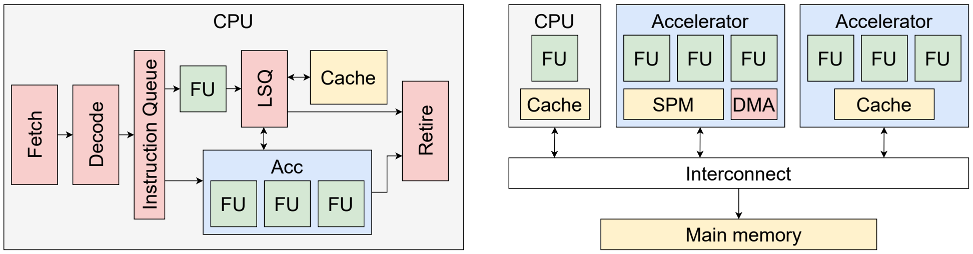
2.1 Domain Specific Accelerators

Limited multi-core scaling in chip multi-processors (CMPs) as a result of the end of Dennard scaling [65], which dictates that transistor power consumption scales with transistor size, has pushed computer architectures into the era of dark silicon where increasing areas of chips need to be turned off as transistor sizes reduce [71]. A large part of a CPU's power consumption comes from instruction overheads. Hameed et al. [102] revealed how functional units in CMPs consume as little as 6% of the total energy when running a video encoding application, highlighting the cost of the generality CPUs provide with instruction fetch/decode and pipelining logic constituting 34% and 22% of the total CPU energy consumption.

In order to improve their performance per unit area, processors have come to incorporate specialized accelerators that fuse commonly used operations into their pipeline, improving data reuse and amortizing instruction fetch/decode costs. While the idea of domain-specific functional units dates back to late 1990s with Intel MMX [141], contemporary in-core accelerators are significantly more complex and narrow in their scope, like Intel Advanced Matrix Extensions [126]. Mobile SoCs are, in fact, no stranger to domain specific acceleration given their limited power and thermal budgets [227, 229].

Accelerators can have varying degrees of coupling with CPUs [58], ranging from *tightly-coupled accelerators* (TCAs, Figure 2.1a) that behave and appear as a functional unit [55, 92, 123, 126, 220], to *loosely-coupled accelerators* (LCAs, Figure 2.1b) that sit outside the CPU core and appear as programmable I/O devices to the operating system (OS) [2, 34, 46, 57, 76, 150, 245]. TCAs share the CPU's memory hierarchy and are programmed using instruction set architecture (ISA) extensions. This allows for easier programming of the accelerator and efficient data sharing, at the expense of complex CPU and ISA verification.

LCAs, on the other hand, are programmed using memory-mapped registers



(a) Tightly-coupled accelerators

(b) Loosely-coupled accelerators

Figure 2.1: Different types of accelerator coupling.

(MMRs) and share data with the CPU via the main memory, with variable support for virtual memory and address translation [57, 200]. By separating the design of the CPU and the accelerators, SoC designers can combine any combination of the two to achieve different performance targets [229]. Such a coupling comes at the cost of invocation [2, 196, 301] and data movement overheads [63, 196, 278, 281], however. Invocation overheads and data movement overheads, specifically pertaining to inter-accelerator data movement, are discussed further in following sections.

2.1.1 Accelerator Programming

The high invocation latency of LCAs has received significant attention over the past two decades as such architectures have become increasingly more important. Perhaps the most well known solution to the problem, one that is used in contemporary architectures, is the use of user-space command queues. Defined in Heterogeneous System Architecture (HSA) [157] specification as Command Queues and in CUDA as Streams [231], user-space command queues are ring buffers in an application's address space that are used to submit task descriptors directly to the accelerator using regular stores. These buffers are mapped into the application's address space via a system call into the accelerator driver, which also notifies the accelerator of the queue's physical location, thus enabling direct communication. Prior work has shown how the hardware's ability to monitor only a fixed number of queues at any given time can be a performance bottleneck [219],

Prior work has also extensively explored additions to ISAs to provide a TCA-like interface to LCAs. Wang et al. [277] propose the EXOCHI architecture that eases the integration and programming of LCAs. EXOCHI abstract accelerators away from host CPUs using an Exoskeleton (EXO) sequencer that is responsible for scheduling application threads onto the accelerator. EXO-sequencers must conform to IA-32 [127] inter-sequencer communication defined earlier in Multiple Instruction Stream Processor (MISP) [104] architecture. This allows the host processor to move a thread context to the sequencer using an instruction called `SIGNAL`, and sequencer to send interrupts to the host for address translation and exception handling (e.g., kernel completion). In order to ease programming of accelerators, the authors also propose C for Heterogeneous Integration (CHI), an OpenMP based framework for writing heterogeneous code that abstracts away host/acceleration interactions (e.g., issuing `SIGNAL` instruction). Pangaea [285] is a CPU-GPU heterogeneous SoC that builds on top of EXOCHI and introduces user-space interrupts and ISA extensions for efficient host-side handling of user-defined events (e.g., write to shared memory by the accelerator).

IBM Wire Speed Processor [76], and its subsequent implementation as PowerEN [34, 150], introduced the `icswx` instruction, an unprivileged instruction that is used to submit computation requests from host cores to accelerators. These requests are encoded into a coprocessor request block (CRB), a cache-aligned 64-byte block that includes input and output data addresses, the computation to be performed, and any additional arguments. IBM POWER9 and z15 processors incorporate a data compression accelerator, called NXU [2], that is also accessible using an unprivileged instruction called `DFLTCC`. While POWER9 uses a longer 128-byte CRB to submit jobs with `DFLTCC`, z15 uses general purpose registers to encode accelerator kernel parameters.

Modern Intel Xeon processors have come to integrate several accelerators [301], like Intel Data Streaming Accelerator [123], that are programmed using a dedicated set of instructions defined under Accelerator interfacing Architecture (AiA). AiA includes instructions to configure the MMIO registers for private accelerators (MOVDIRI/MOVDIR64B), enqueue requests to shared accelerators (ENQCMD/ENQCMDS), and monitor/wait for task completion (UMONITOR/UMWAIT).

2.1.2 Accelerator Chaining

Applications are often composed of several kernels that can be accelerated on modern systems. This makes hardware support for accelerator chaining lucrative since it can greatly minimize (or even completely eliminate) the need to move data through the main memory, avoiding CPU intervention and main memory bandwidth bottlenecks, especially under contention. Such a trend exists for both datacenter-scale big data workloads [90] as well as edge uses like mobile workloads [196] and self-driving cars [67, 170]. Examples of chaining hardware that can *forward* data between accelerators include:

- PCIe peer-to-peer DMA (P2PDMA) [172, 240] allows devices under the same PCIe root complex to access each other's memory using DMA engines, without copying through main memory. P2PDMA works by moving data between PCIe BAR (base address register) regions, regions of device memory exposed into the CPU virtual address space by device drivers. While the size of each region has traditionally been limited to 256 MB, PCIe resizable-BAR enables arbitrarily large BAR sizes, allowing entire device memories to be exposed [20].

- ARM AXI-Stream [24] protocol enables direct communication between on-chip devices connected over ARM AMBA interconnect via a READY/VALID handshake protocol. This can potentially include intermediate buffers to minimize stalls, similar to "flow buffers" for audio/video applications [298], and can be implemented over a crossbar interconnect, enabling parallel communication between multiple unique producer/consumer pairs.

- FUSION [152] is a timestamp-based and lease-based cache coherence protocol for accelerators that optimizes for data movement between cache-based accelerators. FUSION allows the producer to proactively push data from its private cache to the consumer's cache, along with the associated lease, without coherence directory communication. By minimizing the number of hops, FUSION can improve performance compared to non-coherent (i.e., DMA based) and traditional fully-coherent accelerator architectures [88].

- Memory controller optimizations like accelerator-to-accelerator short-circuiting [298] perform store-to-load forwarding inside the memory controller queues where the store and load requests originate from the producer and consumer accelerators, respectively. This builds on the intuition that the producer's stores might still be pending in the memory controller queue when the consumer starts executing. Such forwarding can not only reduce memory access latency, but also reduce the number of precharges and activates. Such kind of forwarding can be combined with memory controller scheduling policies like DASH [271] and FLOSS [304] that optimize bandwidth allocation to the main memory in heterogeneous architectures based on the requesting applications' quality of service requirements.

Designing hardware support for accelerator chaining is only part of the solution, however. Programming for and utilizing such hardware is a key challenge. State-of-the-art systems put this responsibility on programmers, requiring that the application wait for all communicating accelerators to be made available before making an API call that performs the actual transfer. This hurts application performance since there is no way of knowing when a consumer accelerator might be available and, thus, if forwarding would be beneficial at all. Furthermore, stalling the producer kernel's context can prevent the accelerator from being used by other applications, hurting system utilization and fairness. Listing 2.1 highlights this tradeoff in CUDA, where data produced from `gpu0` needs to be forwarded to `gpu1` for consumption. Specifically, the programmer must make a choice based on the allocation on line 8 whether they want to write intermediate results to CPU or wait for `gpu1` to be available. While it is possible to allocate space on both GPUs from the get go, it would lead to wasted memory utilization.

Listing 2.1: Example of CUDA peer-to-peer (P2P) copy, highlighting the need for a context on both the producer *and* the consumer GPU before a copy can be initiated.

```
1  int *gpu0, *gpu1;
2
3  assert(cudaMalloc((void**)&gpu0, sizeof(int) * ARRAY_SIZE) ==
       cudaSuccess);
4  cudaSetDevice(0);
5  producerOnGPU0<<<blocks, threads>>>(gpu0);
6
7  /* Attempt memory allocation on gpu1 */
8  if (cudaMalloc((void**)&gpu1, sizeof(int) * ARRAY_SIZE) ==
       cudaSuccess) {
9      /* Allocation successful.
10      * Now we can directly copy data from gpu0 to gpu1
11      */
12     cudaMemcpy(gpu1, gpu0, sizeof(int) * ARRAY_SIZE,
13                cudaMemcpyDeviceToDevice);
14     cudaFree(gpu0);
15 } else {
16     /* Allocation failed. Now there are two options while we
17      * wait for space on gpu1:
18      * 1. Copy data to cpu and then copy to gpu1
19      * 2. Keep data on gpu0 and wait to forward
20      */
21 }
22
23 cudaSetDevice(1);
24 consumerOnGPU1<<<blocks, threads>>>(gpu1);
```

Solutions to ease forwarding include GAM+ [57], a hardware block that virtualizes accelerators and abstracts away accelerator management from the programmer, requiring the application to send in only a task descriptor of the kernel it wants to execute (e.g., 3D FFT). The requested kernel is then broken down into individual accelerator requests based on the hardware capabilities (e.g., 3D FFT can be expressed using 2D FFTs) and the input size. The accelerator requests are then scheduled as accelerator resources become available, utilizing P2PDMA like semantics to move data between producer/consumer accelerators whenever possible. VIP [196] is another accelerator virtualization framework that abstracts away accelerator management from the programmer. VIP modifies software libraries and drivers to enable applications to schedule a burst of requests on a chain of accelerators. This information is communicated to each accelerator in the chain, which are modified to incorporate buffers that hold context information for each chain they are a part of, along with a scheduler that chooses which chain to process the next request from. Combined with AXI-Stream like buffers, this enables seamless pipelining of accelerators without any host intervention. A key difference between GAM+ and VIP is that the latter guarantees forwarding of data while the former utilizes it opportunistically. In addition, accelerator scheduling is centralized in GAM+ while VIP adopts a distributed scheme. Cohort [281] provides a queue-based interface for CPU/accelerator and accelerator/accelerator communication. Cohort supplements accelerators with a Cohort Engine that can read from/write to software-defined queues in main memory. These queues are created by programmers when defining accelerator chains, and are communicated to Cohort Engines via MMRs configured by device drivers.

Another challenge in chaining accelerators is the reorganization of data between the producer and the consumer accelerators. This might especially be needed for applications in datacenters that combine accelerators from different domains [278]. Data motion accelerator (DMX) [278] is an accelerator that can perform on-the-fly data reorganization, eliminating the need it to be performed by the CPU. Not all systems suffer from this problem, however. SoCs are often tightly integrated with accelerators that are designed to be interoperable [236].

2.2 Processing in Memory

The idea of processing in memory (PIM) dates back to the 1970s [259], but it has been hard to realize due to difficulties in manufacturing high density DRAM and logic on the same die [149]. Recent trends in both workload characteristics and system bottlenecks have pushed the idea to forefront in the past decade, with industrial prototypes indicating that manufacturing issues have at least partially been surmounted [156, 163, 270].

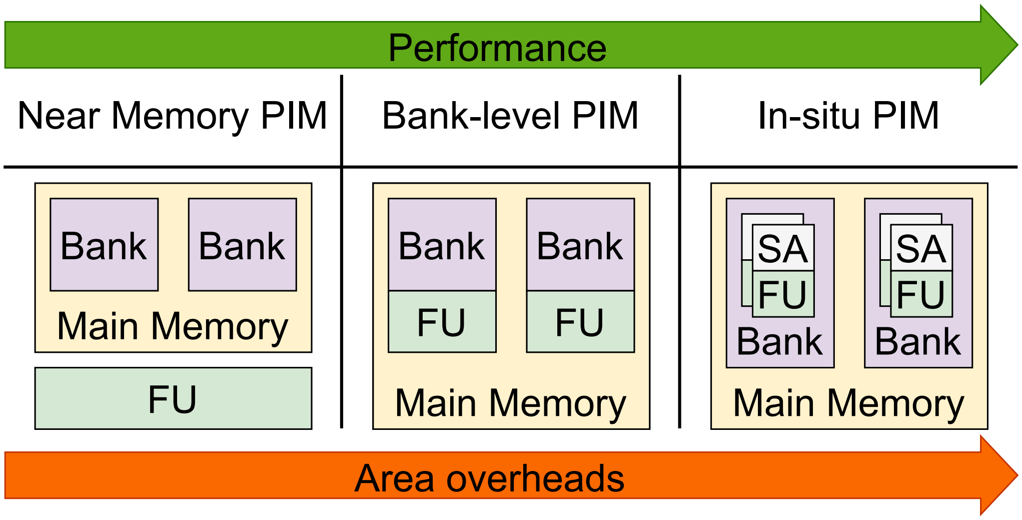
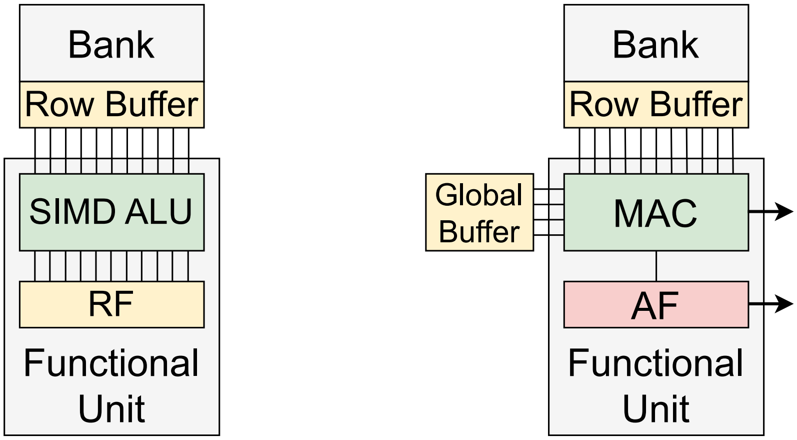


Figure 2.2: Performance/overhead spectrum of processing in memory.

The placement of functional units (FUs) is a key determinant of the performance and area overheads of adding PIM. Figure 2.2 shows three broad categories of PIM architectures. At the lower end of performance/area spectrum is the placement of functional units closer to, but still outside of, memory chips than traditional architectures (e.g., on a logic layer inside 3D stacked memory) [9, 26, 33, 48, 73, 146, 215, 290]. Being outside of the memory cells affords great flexibility in terms of the FU architecture, up to and including traditional processors [9, 26, 33, 215] whose performance is limited largely by thermal constraints only [70]. But, these architectures also offer the least speedup since the compute/memory interface is still limited by the width of the memory bus.

The next step on the spectrum is bank-level PIM where FUs are placed inside each memory bank, usually after the column decoder [3, 108, 145, 155, 156, 161, 163, 164]. While being significantly more area constrained, such architectures benefit from significantly reduced data movement latency and energy by avoiding communication over the memory bus. Moreover, computing across all banks achieves wide data parallelism, with some architectures supporting lock-step PIM operations across multiple banks [3, 108, 155, 156, 161, 163] to improve performance. The latter, in particular, has been the basis for commercial implementations from SK Hynix [108, 155, 163] and Samsung [156, 161], shown in Figure 2.3.



(a) Samsung

(b) SK Hynix

Figure 2.3: High level architecture of bank-level PIM prototypes built by Samsung (a) and SK Hynix (b). RF: register file, MAC: multiply and accumulate, AF: activation function.

Samsung’s design, based on HBM, incorporates a functional unit at each bank that can perform a row buffer wide SIMD computation on data from the bank row buffer and local register file (RF) (Figure 2.3a). The results of the computation can be written back to the bank and then retrieved by the host using regular loads. SK Hynix's design, meanwhile, is based on GDDR6 and is tailored for machine learning algorithms, incorporating a multiply-and-accumulate (MAC) unit that is fed from the bank row buffer and from a global buffer that is shared across all banks (Figure 2.3b). This result can optionally be passed through another ALU that implements an activation function (AF, e.g., ReLU). Both the results are stored in temporary registers that can be read using dedicated instructions.

At the high end of the PIM performance/area spectrum is the placement of function units inside memory subarrays, called in-situ PIM [18, 250, 256, 286], offering extremely wide data parallelism (e.g., 8 KB across a rank [250]) and further reductions in data movement energy. These benefits incur substantial area and complexity overheads, however. Recent work suggests that MATs inside subarrays are packed significantly more tightly than believed earlier, further propounding the overheads [184].

While the above presented classification categorizes most PIM architectures, there are some exceptions. Charge sharing techniques [84, 249, 250] violate DRAM timing constraints and activate multiple rows within a subarray to perform bulk bit-wise operations like copy [84, 249] and logical AND/OR [84, 250]. These techniques modify the memory controller and/or peripheral DRAM circuitry, but not the memory cells themselves. UPMEM [99, 270] is a PIM architecture that integrates a RISC-style fine-grained multithreaded processor, called a data processing unit (DPU), at each memory bank. Each DPU has a local instruction and data scratchpad memory, and communicates with the memory bank using a DMA engine. While similar to the bank-level PIM architectures discussed before, UPMEM architecture is significantly more complex and differs in that it does not perform computation at the row buffer itself. Rather, DPUs appear as another host to the memory banks. Moreoever, DPUs are independent Turing-complete processors that are not synchronized to DRAM timing parameters, unlike other bank-level PIM architectures.

2.2.1 PIM Programming

PIM offloading can be broadly categorized as either coarse grained or fine grained [197]. Coarse grained offloading involves the invocation of an entire kernel (e.g., streamcluster [48]) on oft complex PIM units like course grained reconfigurable arrays (CGRAs) [73] and traditional processors [9, 26, 270]. These PIM units can be programmed through MMRs [9, 32, 48, 73] or ISA extensions [26], entailing significant modifications to the memory controller to perform PIM computation while handling non-PIM requests from the host [32, 48, 73].

Fine grained offloading, on the other hand, involves the host submitting requests that are significantly more elementary in their computation (e.g., vector addition [161]), similar to CPU instructions. Given the nature of these PIM instructions, the PIM units tend to be simpler in their complexity and often adhere to DRAM timing constraints [7, 108, 161, 198]. This not only makes the architecture more flexible, but also simplifies memory controller side logic since the host is responsible for injecting PIM instructions [197]. While the host is traditionally also responsible for correctly ordering PIM instructions, Nag et al. [197] show how traditional memory barriers are neither necessary nor sufficient for correctly ordering PIM requests. They propose Orderlight, a lightweight memory barrier that prevents reordering of PIM requests all the way from the host processor to the memory controller. Unlike traditional fences that stall the processor pipeline until prior memory accesses are globally visible, Orderlight barriers require each component that the PIM instructions pass through to maintain ordering. This is done by issuing the barriers to the memory, alongside loads, stores, and PIM requests, allowing the interconnect and memory controller to be aware of the relative ordering of PIM requests.

Maintaining coherence between host and PIM units is another key challenge. While some architectures leave this responsibility to the programmer [108, 161] or simply mark PIM memory as uncacheable [9, 48, 198], several optimizations have been proposed. PIM-Enabled Instructions (PEI) [7] is a locality-aware PIM offloading framework that executes the instructions either on the host or on PIM based on the location of its input operands. LazyPIM [32] is a PIM-aware coherence protocol that is inspired from transactional memory. PIM kernels speculatively performs computation assuming exclusive access to data and submit the set of read and write addresses accessed to the host CPU. If the CPU detects a conflict in accesses (e.g., PIM read a CPU written address), the CPU flushes the conflict lines and restarts the PIM kernel.

2.3 Emerging Memory Technologies

2.3.1 Intel Optane

Intel introduced Optane Data Center Persistent Memory Module (DCPMM, simply referred to as Optane in this dissertation) in 2019, a PCM-based 3D Xpoint memory technology that is significantly more dense than traditional DRAM technology [121, 50]. Packaged in a DIMM form factor, Optane is byte-addressable and fits into regular DDR4 slots, communicating using a custom DDR4-based protocol called DDR-T [122, 307]. While extremely high capacity and energy efficient, Optane suffers from worse performance and limited write endurance, all properties of the underlying material. Prior work has shown that Optane achieves nearly 2.5x lower sequential read bandwidth compared to DRAM and about 6x lower write bandwidth [129, 218, 293]. These evaluations show a non-linear relationship between increasing concurrency and write bandwidth. Being PCM-based also limits the life of each memory module in terms of its write endurance [121].

Optane can be configured in two modes: App Direct and Memory. App Direct mode exposes Optane as a fast storage device with an optimized file system that bypasses the Linux page cache, enabled by its byte-addressable capabilities [21, 291]. Memory mode, meanwhile, exposes it as large main memory with DRAM serving as a direct-mapped cache to hide its performance deficiencies. Within App Direct mode, it is also possible to interface with the memory directly without any file system. This allows libraries like Memkind to expose Optane as a memory-only NUMA node, creating a flat memory hierarchy between DRAM and Optane and enabling direct use of the large memory pool by applications [30]. We use this support to compare DRAM and Optane performance.

In its Q2 2022 financial report, Intel announced that it is closing its SSD business, including Optane, and writing off any leftover inventory [124]. This was followed by a technology brief detailing a shift in strategy to CXL-attached memory [125]. While no longer in production, Optane continues to serve as an effective evaluation platform for high capacity byte addressable memory technologies [68, 143]. Furthermore, CXL-expanded memory can be backed by SSDs [125, 140, 216, 243, 295] and even Optane itself [125], ensuring broader applicability of findings related to Optane.

2.3.2 Compute Express Link

Compute express link (CXL) [60] was announced in 2019 [37] as an industry-standard interconnect technology to connect processors, devices, and memory expanders over the PCI Express bus interface. The CXL standard defines three protocols: CXL.io for I/O devices, CXL.cache for cache-coherent devices (e.g., caching accelerators), and CXL.memory for memory-enabled devices (e.g., memory expanders). The CXL 1.1 specification [61] defines three types of devices and what combination of the three protocols each would use.

Of the three types of CXL devices, Type-3 is of particular note. Defined for memory expanders, Type-3 devices use CXL.io and CXL.memory protocols to allow for coherent expansion of main memory capacity over PCIe. By providing load/store semantics similar to traditional main memory, Type-3 devices (simply referred to as CXL memory henceforth) provide transparent expansion of main memory without the limitations of traditional DDR interfaces. Compared to DDR, PCIe offers higher per-pin bandwidth and lower per-bit transfer energy [262]. Moreover, the memory technology across the interconnect is not bound to be DRAM, allowing for use of high density media like SSDs [125, 140, 216, 243, 295] and Optane [125], as shown in Figure 2.4.

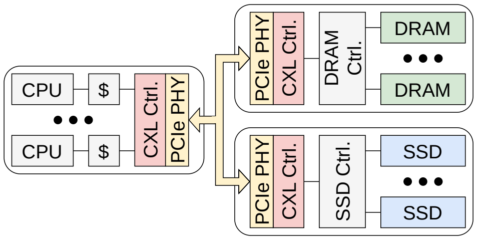


Figure 2.4: CXL Type 3 devices enable transparent, coherent, and technology agnostic expansion of main memory.

Like Intel Optane, CXL memory capacity comes at a performance cost. While performance varies with the design of the CXL controller itself [262], CXL adds at least 70 nanoseconds to round-trip memory access latency [252], not accounting for contention or memory technology variations at the expander. The achievable bandwidth is also limited by the underlying PCIe technology standard, which is 64 GB/s for the latest PCIe 5.0 x16 link [284]. In comparison, our DDR4-based evaluation system in Chapter 5 achieves 157 GB/s across 8 memory channels. While PCIe 6.0 nearly doubles this bandwidth to 121 GB/s, it has not yet entered production at the time of writing.

2.4 GPU/Host Communication

GPUs predominantly use a push-based communication paradigm where the programmer is responsible for moving data up-front using DMA engines [16, 203]. This “copy then execute” model puts a significant burden on the programmer when it comes to minimizing data movement overheads, such as by overlapping it with computation in a pipelined fashion.

Nvidia introduced Unified Virtual Addressing (UVA) [247] in 2011 to ease this burden by integrating GPU memory into CPU's address space. This enables passing of CPU pointers to GPU kernels for direct "zero-copy" access and makes it possible to move data between GPUs without going through CPU bounce buffers. Building on top of UVA, Unified Memory (UM) [107, 203] provides the illusion of a single large coherent memory pool consisting of CPU and GPU memory that requires no programmer-initiated copies. Unlike UVA, UM transparently moves data between CPU and GPU memory at a page granularity on-demand, serving as a replacement for the traditional "copy then execute" model. More importantly, however, is the fact that UM allows for GPU memory "oversubscription" wherein kernels can create allocations larger than GPU memory size. This is especially useful for machine learning (ML) applications and is supported by major ML frameworks [1, 51] and GPU simulators [174].

Unified memory's convenience comes at the cost of performance, however. The cost of servicing a single page fault is in the range of 20-45μs [79, 310] and is largely dominated by software overheads [11]. These properties limit the use of UVM in high-performance scenarios.

2.5 Large Language Models

The advent of modern LLMs is largely attributable to the Transformer [272], a machine learning network architecture that is able to extract word meanings and contextualize them as part of the broader prompt. Figure 2.5 shows the architecture of a decoder-only transformer model and the auto-regressive nature of LLM inference. At the heart of a decoder block is the multi-head attention (MHA) layer. The symbols Q, K, and V represent the query, key, and value vector representation of each token, obtained by multiplying the embedding vector of each token with the respective weight matrices. Each self attention block, called an attention *head*, extracts different semantic meaning from each token based on its weights. The decoder concatenates the output of each head with the original vector representation of each token to update its meaning and then passes it through a feed forward network (FFN) layer, often implemented as a multi-layer perceptron. FFN layer further refines the meaning of each token based on learned weights, producing another delta vector that is used to update each token.

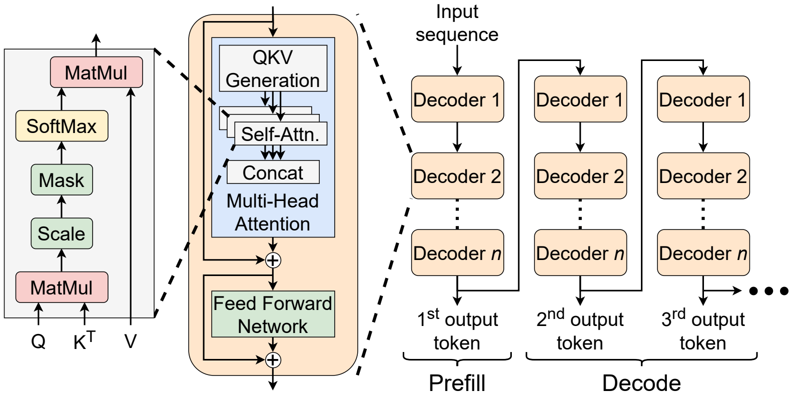


Figure 2.5: Architecture of a decoder-only LLM. Prefill and decode perform GEMM and GEMV operations, rendering them compute and memory bound, respectively.

Intuitively, the query representation of a token can be imagined to be asking the question “Are any tokens before me relevant to me”, while the value representation of each relevant token would be answering “Yes, I am!”[1] When the query representation of a token is multiplied with the key representation of each token (vector dot product), the result is a value in the range $(-\infty, +\infty)$, with higher values representing stronger correlation. Passing this result vector through softmax returns a probability distribution in the range [0, 1], signifying the relative importance of each other token in the sentence to the token under consideration. Finally, each of these probability values are multiplied with the value representation of the corresponding token and summed up, providing us with a *delta* vector. This delta vector, when added to the token embedding, modifies the meaning of the token in a way that is (hopefully) more refined and contextually richer. Repeating this operation across multiple heads in parallel, each with its own set

[1] Examples and intuitive explanations in this section are heavily based on Grant Sanderson’s wonderful visual explainer series [244].

of query, key, and value weights, allows for different semantic interpretations of the input to influence the representation of each word.

Figure 2.5 also highlights two distinct inference phases: prefill and decode. During prefill, the entire input sequence is parsed by the model in a series of GEMM computations to produce the first token and store the key/value (KV) representations in a KV cache. In successive iterations, the decode stages utilize the KV cache and parse only the previously generated token in a series of GEMV computations to predict the next token. Note that the initial prompt is not parsed in its entirety because the final representation of each token is dependent on the preceding tokens *only*. This is why the KV cache stored earlier is useful, since it avoids redundant computation for the already parsed tokens.

Because of differences in the operational intensity of GEMM and GEMV, prefill is usually compute-bound while decode is memory-bound. Since most generative AI applications produce thousands of output tokens (e.g., LLaMa 4 has a context window size of 10 million tokens [190]), LLM inference is largely memory bound. A common technique to improve data reuse is batching of multiple requests, effectively turning the GEMV computations in decode phase to GEMM computations. The size of each batch is, however, limited by the storage requirements of the KV cache for each prompt.

Memory performance is not the only limiting factor, however. Model sizes since the introduction of transformers have exploded in size, from 175 billion in GPT-3 in 2020 [35] to 2 trillion in LLaMa 4 in 2025 [190]. Larger models incorporate more attention heads and larger weight matrices, allowing for semantically richer understanding of text and improved model performance. This exponential growth, however, has far outpaced the growth in AI hardware memory capacity as well [87], demanding solutions that offer both high bandwidth *and* capacity.

2.6 PIM on Emerging Memory Technologies

Given the memory capacity and performance demands of applications like LLMs (Section 2.5), recent work has looked at putting PIM (Section 2.2) on high capacity memory technologies (Section 2.3).

PIM on CXL devices is particularly attractive because CXL devices are not as area and power restricted as traditional DDR memory. Memory-Mapped Near-Data Processing (M^2NDP) [101] extends Type-3 CXL devices (Section 2.3.2) with a RISC-V [25] based fine grained multithreaded processor in a near memory PIM fashion (Section 2.2). M^2NDP modifies the CXL device to expose an uncacheable memory region that applications can access using regular CXL.mem reads/writes to register PIM kernels, launch them, or check the status of their execution using a predefined ISA. CENT [93] combines near memory PIM and bank level PIM (Section 2.2) on Type-3 CXL devices to distribute the processing of LLM decoder blocks (Section 2.5) on the compute unit most suited for each kernel. Specifically, the bank-level PIM architecture is based on GDDR6-AiM [108, 155, 163] and is optimized for large data kernels like GEMV. The near memory PIM units, meanwhile, are composed of RISC-V cores and accelerators that compute smaller kernels like softmax. Communication between the two PIM units is performed using a shared buffer, and they are programmed by writing instructions from the host to a memory-mapped instruction buffer. The two PIM units are programmed by writing instructions from the host to a memory-mapped instruction buffer. Intra-device communication is handled using an on-chip shared buffer while inter-device communication is performed via CXL.mem reads/writes to this shared buffer.

MAGIC [153] is a memristor [54, 260] based PIM architecture that performs sub-array level NOR computation in memristor crossbar arrays (used in non-volatile ReRAM [45]) by manipulating column voltages. MPIM [120] is another memristor based PIM architecture for low power edge devices that exploits analog timing characteristics of memristors to perform bank-level nearest neighbor search. It also supports logical bit-wise computations via additional sense amplifiers. Resch et al. [232] demonstrate that STT-MRAM [19, 223] based computation is idempotent, i.e., any operation repeated multiple times produces the same result. They utilize this property to construct MOUSE, a PIM architecture for energy harvesting intermittent computing devices that supports bank-level bit-wise computations. Resch et al. [233] evaluate the endurance impact of integrating in-situ PIM (Section 2.2) into byte-addressable nonvolatile memory. Their analysis reveals how PIM can wear down a 1024x1024 STT-MRAM memory array in a few days, with cells holding temporary data wearing down faster due to an imbalance in write frequency. Even with several software and hardware schemes to balance writes across the memory, endurance improves by up to 2.22x only, highlighting the need for better memory/PIM codesign to make such architectures a reality.

3 Relieving Memory Pressure In SoCs Via Data Movement-Aware Accelerator Scheduling

Modern smartphone systems-on-chip (SoCs) comprise of several dozens of domain-specific hardware accelerators dedicated to processing audio, video, and sensor data [227]. These accelerators, which sit outside the CPU pipeline, appear as programmable I/O devices to the OS and communicate with the CPU using memory-mapped registers and shared main memory, sometimes connecting to the last-level cache [58]. To maximize performance and accelerator-level parallelism [228], applications can request a chain of accelerators running in producer/consumer fashion [196]. The speedups these chains provide, however, is limited by the fact that the accelerators communicate via the main memory, creating contention at the memory controller and the interconnect. This bottleneck will worsen as SoCs become more heterogeneous and incorporate accelerators for more elementary operations [57].

Techniques to reduce this contention include 1) *forwarding* data from the producer to the consumer, i.e., moving data from the producer's local memory directly to the consumer's, and 2) *colocation* of consumer tasks with producer tasks, thus eliminating all data movement. Examples of *forwarding* techniques include insertion of intermediate buffers between producer and consumer accelerators (VIP [196, 298]) or optimizing the cache coherence protocol to proactively move data from producer's cache to the consumer's cache directly (FUSION [152]). The former requires design-time determination of communicating accelerator pairs, while the latter requires that the accelerators use caches and be part of the same coherence domain, limiting their scalability and flexibility. More recent techniques include ARM AXI-stream [17, 24], which allows multiple producer/consumer buffers to be connected over a crossbar switch, and Linux P2PDMA [172, 240], which enables direct DMA transfers between PCIe devices without intermediate main memory accesses. Unlike VIP and FUSION, they allow for dynamic creation of producer/consumer pairs at run time in order to move data between them. Efficient utilization of such *forwarding* techniques, however, remains a challenge.

Existing systems expect software to explicitly utilize the forwarding mechanism to move data between producer and consumer [183, 196, 203], requiring knowledge of task mapping to accelerators. Distributed management of tasks by each accelerator, however, results in the accelerator's inability to utilize forwarding mechanisms due to the lack of knowledge of task mappings to other accelerators. A centralized accelerator manager has a global view of the system, allowing implementation of policies that opportunistically employ forwarding mechanisms to improve accelerator utilization and application performance. Unfortunately, the scheduling policies employed thus far [57, 78] by these managers are not designed to efficiently utilize forwarding hardware.

Scheduling policies typically prioritize tasks using arrival time, deadline, or laxity. Such policies can be extended to prioritize tasks that may forward data from a producer, similar to FR-FCFS scheduling in memory systems [235], where row buffer hits are prioritized over older tasks. However, this can lead to unfairness where an application with more forwards can starve others with fewer forwards. Therefore, **we need a scheduling policy that can opportunistically perform data forwards while still providing fairness and quality of service (QoS)**.

In this chapter, we introduce RELIEF, an online accelerator scheduling policy that has forwarding, QoS, and fairness as first-class design principles. RELIEF prioritizes newly ready tasks over existing ones since they can move data directly from the producer's memory using forwarding mechanisms. RELIEF provides QoS in terms of meeting task deadlines and fairness in terms of reducing variance in application slowdown due to contention. It achieves both by tracking task laxity and throttling priority elevations if they can cause missed deadlines. These properties matter where tail-latency is important, such as user-in-the-loop smartphone and client-server applications. We evaluate RELIEF on a suite of vision and machine learning benchmarks with strict latency constraints on a mobile platform. Our key contributions are:

- An evaluation of data movement overheads for low-latency accelerator chains used in deadline-constrained vision and machine learning applications. We observe that some of these applications spend as much as 75% of their execution time on data movement.
- A novel scheduling policy, called RELIEF, that maximizes utilization of existing forwarding hardware. RELIEF can be easily integrated into existing hardware managers and is agnostic of both the forwarding mechanism and the specific definition of laxity, allowing for wider adoption.
- Extensive evaluation of RELIEF on a simulated mobile SoC, encompassing performance improvements, implementation overheads, and sensitivity to microarchitectural design decisions. RELIEF achieves up to 50% more forwards compared to state-of-the-art (SOTA) policies, resulting in 32% and 18% lower main memory traffic and energy consumption, respectively. Simultaneously, RELIEF improves QoS by meeting 14% more task deadlines on average, and improves fairness by reducing the worst-case deadline violation by 14%.

3.1 Background

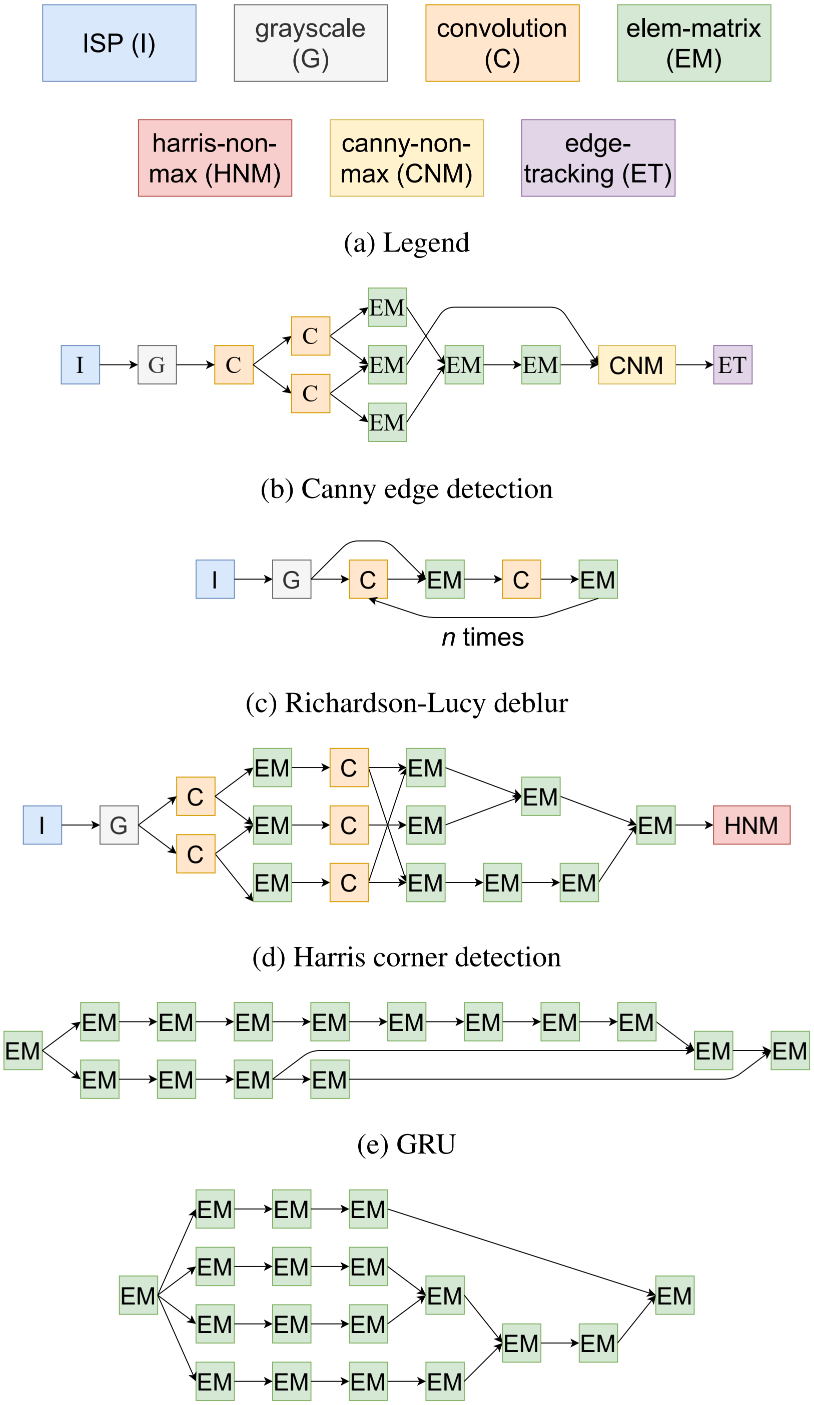
General-purpose processors and domain-specific accelerators represent two ends of a spectrum of performance and flexibility, with the latter trading off the former's versatility for improved performance. A middle ground between the two approaches is to have a set of accelerators for elementary operations that can be stitched together dynamically by each application to serve its needs [57]. This is supported by the observation that applications across domains are often composed of similar kernels [92]. Such an approach eliminates redundancy of hardware functional units along with greatly minimizing the need for a specialized accelerator for each new application.

In this section, we present a suite of real-time smartphone workloads that are widely used in modern devices and discuss how they can be broken down into a set of elementary accelerators. We quantify how memory-bound these accelerators are, motivating the need for techniques to reduce data movement costs. Next, we discuss the functionality of an accelerator manager [57] and why they are well-equipped to improve hardware utilization and provide QoS. Finally, we present examples to explain how SOTA accelerator scheduling policies fall short in utilizing forwarding hardware.

3.1.1 Modern smartphone workloads

We study two important classes of modern smartphone workloads in this chapter: vision and recurrent neural networks. Both classes together represent a wide variety of compute-intensive user-facing applications, making them suitable for hardware acceleration.

Computer vision: Mobile visual computing applications have exploded in popularity in recent years, ranging from complex photography algorithms to AR/VR applications [74]. These applications often utilize several common image processing kernels. One example is *Canny edge detection* [38], which is used in face detection, both alone [181] and as part of neural network pipelines [257]. Another example is *Har-*



(a) Legend

(b) Canny edge detection

(c) Richardson-Lucy deblur

(d) Harris corner detection

(e) GRU

(f) LSTM

Figure 3.1: Kernels in different image processing and RNN applications.

ris corner detection [105], which is used for feature extraction in panorama stitching algorithms [168], especially in VR applications [173]. *Richardson-Lucy deconvolution* [178, 234] is an image deblurring algorithm that sharpens shaky camera images. These three applications are commonly fed images directly from an image signal processor (ISP) that captures raw camera output and performs preprocessing operations like demosaicing, color correction, and gamma correction [109].

Recurrent neural networks (RNNs): These are a class of machine learning kernels used for time-series data, wherein the inference at a time step can affect the inference at a later time. This makes them particularly well-suited for speech recognition [226] and language translation [263] applications in modern phones. We evaluate two different RNN applications: *long short-term memory (LSTM)* [114] and *gated recurrent unit (GRU)* [49]. Given their widespread use, RNNs have been the subject of prior work in low-latency accelerator design [85] and accelerator scheduling [53, 299].

Details about these benchmarks, including their deadline and input size, are listed in Table 3.5. These applications can be represented as directed acyclic graphs (DAGs) of seven compute kernels, each of which can be implemented as a separate hardware accelerator, as shown in Figure 3.1. The description of each accelerator is listed in Table 3.1. These accelerators are ultra low-latency, spending significant time moving data to/from memory. The data movement overhead for each accelerator and each application is quantified in Table 3.2. For each application, the table compares the memory time without forwarding hardware to an ideal scenario where forwarding hardware is used whenever possible.

The percentage of time spent on data movement by each accelerator is primarily a function of its operational intensity. Accelerators like `convolution` have abundant data reuse, which leads to high operational intensity and a higher compute-to-memory access time ratio. Meanwhile, `elem-matrix` has little to no data reuse depending on the operation requested, which causes its run time to be dominated by memory access latency. The frequency of use of each accelerator type dictates how much time

Table 3.1: Elementary accelerators used in this work. SPAD stands for local scratchpad memory.

Accelerator (SPAD size in B)	**Description**
canny-non-max (262,144)	Suppress pixels that likely don't belong to edges.
convolution (196,708)	Convolution with a max. filter size of 5x5.
edge-tracking (98,432)	Mark and boost edge pixels based on a threshold.
elem-matrix (262,144)	Element-wise matrix operations including add, mult, sqr, sqrt, atan2, tanh, and sigmoid.
grayscale (180,224)	Convert RGB image to grayscale.
harris-non-max (196,608)	Enhance maximal corner values in 3x3 grids and suppress others.
ISP (115,204)	Perform demosaicing, color correction, and gamma correction on raw images.

each application spends on data movement. GRU and LSTM, which exclusively use `elem-matrix`, spend nearly 75% of their run time moving data between accelerators while Deblur, which relies heavily on `convolution`, spends a mere 3%. More importantly, we can see how efficient use of forwarding hardware can significantly reduce data movement overheads, especially for memory heavy RNN applications.

3.1.2 Accelerator manager

The use of dedicated hardware to manage the execution of accelerators frees up the host cores from performing scheduling and serving frequent interrupts from accelerators [57], especially for applications with thousands of low latency nodes. [1] The manager implements a runtime consisting of a host interface, a scheduler, and driver functions for each accelerator type.

[1] We use the terms node and task interchangeably.

Table 3.2: Absolute time spent in compute vs data movement. These are sum totals and do not account for computation/communication overlap.

Accelerator	**Time (us)**	
	Compute	**Memory**
canny-non-max	443.02	30.45
convolution	1545.61	18.25
edge-tracking	324.73	13.56
elem-matrix	10.94	30.44
grayscale	10.26	15.23
harris-non-max	105.01	13.77
ISP	34.88	8.71

Application	**Time (us)**		
	Compute	**Mem (no fwd)**	**Mem (ideal)**
canny	3539.37	237.74	173.29
deblur	15610.58	509.80	420.06
gru	1249.31	3343.72	1608.01
harris	6157.30	372.19	303.16
lstm	1470.02	3879.98	1797.77

Host interface: The CPU and the hardware manager communicate via shared main memory, with user programs submitting tasks to the manager via either a system call or user-space command queues [157, 231].

Scheduler: The submitted tasks are written into queues in the main memory that can be read directly by the hardware manager. The hardware manager performs sorted insertion of these tasks into their respective accelerator's ready queue using a scheduling policy. These policies typically sort using arrival time, deadline, or laxity.

Drivers: Tasks from ready queues are then launched onto accelerators via driver functions. Drivers manipulate accelerators or their DMA engine's memory-mapped registers (MMRs) to launch computations or load/store data, respectively.

Hardware managers can be realized as an accelerator themselves or as a microcontroller, with the latter trading off latency for ease of implementation and flexibility [78].

3.1.3 Limitations of SOTA scheduling policies

To illustrate how contemporary accelerator scheduling policies underutilize forwarding mechanisms, consider the two DAGs presented in Figure 3.2a. The number of each accelerator type available is indicated in the "Accelerators" box. The color inside each node represents the type of resource it requires. The upper number is the execution time of the node while the lower number is the deadline. The node deadlines have been computed using critical-path method assuming both DAGs arrive at time 0 and have deadlines of 16 and 15 time units, respectively.

We compare the schedules generated by four SOTA policies to an ideal schedule. Each of the policies presented below work by sorting a per accelerator-type ready queue based on the described criteria. As an accelerator of a given type becomes available, the manager runtime pops the head of the queue for execution.

1. *First Come First Serve (FCFS)*: Simplest baseline policy where incoming tasks are appended to the tail of the ready queue. FCFS represents the non-preemptive version of round-robin scheduling used in GAM+ [57].

2. *Global Earliest Deadline First (GEDF)*: A straightforward extension of the uniprocessor optimal Earliest Deadline First (EDF) policy, where the tasks are sorted based on increasing deadline. There are two variants depending on how the task deadlines are computed:

    (a) *GEDF-DAG (GEDF-D)*: Uses the deadline of the DAG that the task belongs to as the task deadline. This was previously used in VIP [196].

(b) *GEDF-Node (GEDF-N)*: Sets the task deadline by performing critical-path analysis on the DAG. This is amongst the most well-studied policies in real-time literature [241, 294, 303].

3. *Least Laxity First (LL)*: Another uniprocessor optimal scheme [66], this policy works by sorting tasks in increasing order of their *laxity*, which is defined below in Equation 3.1. The deadline used here is set using the critical-path method.

$$laxity = deadline - runtime - current\_time \tag{3.1}$$

4. *LAX [299]*: A variant of LL that de-prioritizes tasks with a negative laxity in favor of tasks with a non-negative laxity to improve the number of tasks that meet their deadline. We use this variant of LL for comparison in the rest of the chapter.

5. *HetSched [12]*: A least-laxity first policy that assigns task deadlines using the following equation:

$$deadline_{task} = SDR \times deadline_{DAG} \tag{3.2}$$

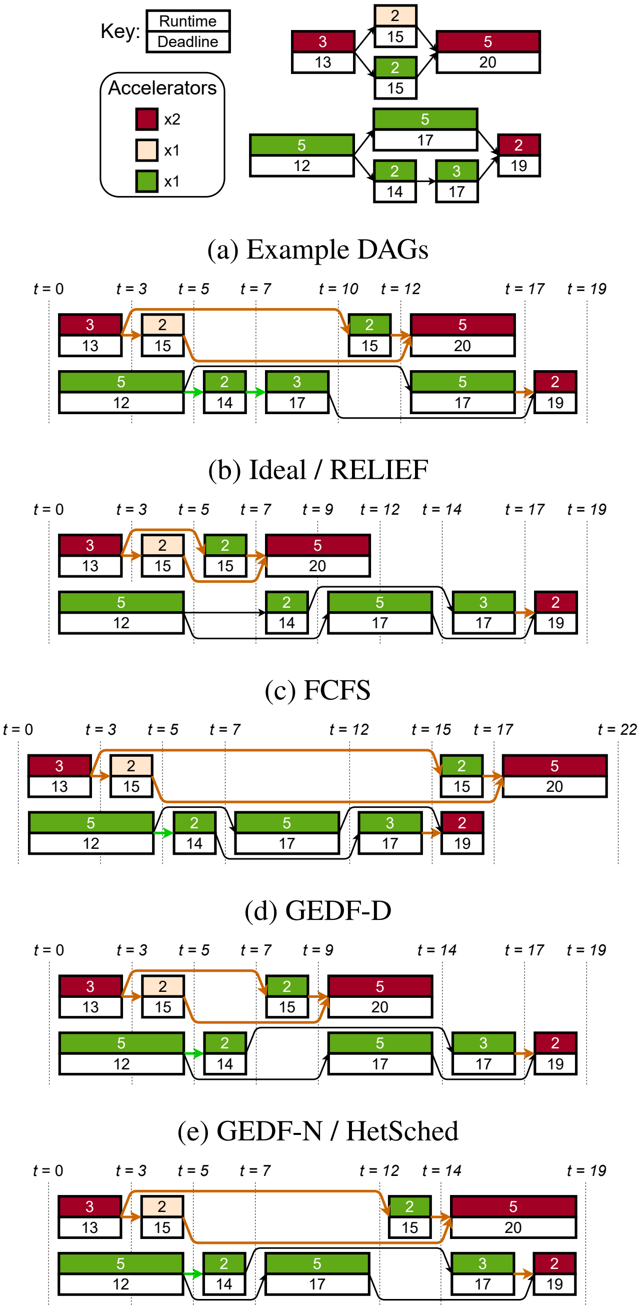
Here, *sub-deadline ratio (SDR)* quantifies the contribution of a task to the execution time of the path it is on.

Figure 3.2 shows the possible schedules generated by each of the policies above. The figures shows both cases, one where data is forwarded from producer to consumer (brown arrow), and another when computation is colocated, putting the consumer computation on the same accelerator as the producer, thereby eliminating all data movement (green arrow). For the same number of forwards, the policy that generates more colocations is therefore the better one. Note that intermediate results are dispensable; we only care about the final output. Looking at the ideal schedule in Figure 3.2b, we observe that it not only meets deadlines, but also achieves 5 forwards and 2 colocations. The ideal policy is able to achieve these forwards and colocations by running the consumer nodes immediately after producer nodes, allowing for better utilization of the aforementioned forwarding techniques. To the best of our knowledge, this is an optimization that no current scheduling policy performs. All other policies, barring GEDF-D, meet the deadline but miss out on forwarding opportunities. FCFS achieves 5 forwards, but performs no colocations. GEDF-D achieves better forwarding with 5 forwards and 1 colocation but misses deadlines. GEDF-N and HetSched produce the same schedule where they meet deadlines but with sub-optimal number of colocations. LAX has a different schedule than GEDF-N/HetSched, but achieves the same number of forwards. We, therefore, need a scheduling policy that exploits forwarding opportunities while being deadline aware.

3.2 RELIEF: Relaxing Least-laxity to Enable Forwarding

3.2.1 Scheduling algorithm

We now present RELIEF, *RElaxing Least-laxIty to Enable Forwarding*, our proposed LL-based policy that attempts to maximize the number of data forwards while delivering QoS. The key idea behind the policy is to promote nodes whose parents have just finished execution, ensuring that the children can forward the data from the producer before it is overwritten. To reduce unfairness and missed deadlines such promotions might cause, RELIEF employs a laxity-driven approach that throttles priority escalations when deadlines could potentially be missed. By combining priority elevations with laxity-driven throttling, RELIEF achieves the ideal schedule shown in Figure 3.2b as well as the ideal data movement time in Table 3.2. We can see from the figure how RELIEF's behavior deviates from LAX, another LL-based policy, at timestep 7, where RELIEF favors the second DAG's newly ready child over existing ready nodes with lower laxity and deadlines.



(a) Example DAGs

(b) Ideal / RELIEF

(c) FCFS

(d) GEDF-D

(e) GEDF-N / HetSched

(f) LAX

Figure 3.2: Comparison of FCFS, GEDF-D, GEDF-N, LAX, and HetSched to an ideal schedule. RELIEF achieves the ideal schedule. Brown and green arrows represent forwarding and colocation, respectively. The “Accelerators” box indicates the available number of each accelerator type.

The RELIEF algorithm is presented in Algorithm 1. Newly ready nodes whose parents have just finished execution are called *forwarding nodes*, since they can potentially forward data from the producer's local memory. RELIEF schedules these forwarding nodes immediately if there are resources available, bypassing nodes with lower laxity if they can meet their deadline under a LL scheme. If no priority escalation is possible, the algorithm proceeds in a vanilla LL fashion. We also experiment with LAX's de-prioritization mechanism that allows tasks with non-negative laxity to bypass those with negative laxity in the ready queue (Section 3.1.3). While this mechanism can improve the number of tasks that complete by their deadline (Section 3.4.4), we show that it can lead to unfairness in Section 3.4.5.

RELIEF works by creating a laxity-sorted list of candidate forwarding nodes, called *fwd_nodes*, from newly ready nodes (Algorithm 1, lines 2-8). We store laxity as *deadline - runtime*, subtracting the current time from it when manipulating the ready queue (Algorithm 2, line 6). The candidate nodes are then inserted into the ready queue at either the front (Algorithm 1, line 17) or at the position dictated by their laxity (Algorithm 1, line 22). A candidate node is escalated in priority only if 1) the number of forwarding nodes in the ready queue for an accelerator type is less than the number of idle instances of that type (controlled by `max_forwards`), and 2) the function `is_feasible()` returns true. The first condition ensures that forwarding nodes are always the next to run, ensuring their input data is still live in its producer's local memory. `is_feasible()` returns true if the priority escalation of the candidate node is unlikely to cause deadline misses. Our evaluation shows that predicting node *runtime* once at the time of insertion into the ready queue has sufficient accuracy (Section 3.4.6).

The key to minimizing missed deadlines is `is_feasible()`'s ability to predict which node promotions might cause them. It takes three arguments: the ready queue, the candidate forwarding node, and its position in the ready queue based on laxity. In our implementation, presented in Algorithm 2, we use a laxity-driven approach. For each node in the ready queue that has a higher priority than the candidate node, we

Algorithm 1: RELIEF

```
Function RELIEF(finishing_node):
    for child $\in$ finishing_node.children do
        child.cmplt_parents += 1
        if child.cmplt_parents == child.num_parents then
            child.runtime = predict_runtime(child)
            child.laxity = child.deadline - child.runtime
            index = find_pos(fwd_nodes[child.acc_id], child)
            fwd_nodes[child.acc_id].insert(index, child)
    for each acc_id do
        max_forwards = num_idle_accelerators[acc_id]

        while not fwd_nodes[acc_id].empty() do
            node = fwd_nodes[acc_id].pop_front()
            index = find_pos(ready_queue[acc_id], node)

            if max_forwards > 0 and is_feasible(ready_queue[acc_id], node,
              index) then
                ready_queue[acc_id].push_front(node)
                node.is_fwd = true
                max_forwards -= 1
                update_fwd_metadata(finishing_node, child)
            else
                ready_queue[acc_id].insert(index, node)
                node.is_fwd = false
```

ensure that its laxity is more than the candidate node's run time. That is, each of those nodes can tolerate the additional latency of the candidate node without missing their deadline. Since the queue is already sorted by laxity, we start at the head of the queue and find the first node that is 1) itself not a forwarding node, and 2) has positive laxity. If the node thus found has laxity greater than the candidate node's runtime, then every following node does too and the candidate node's priority can be safely escalated. The first condition here ensures that existing forwarding nodes do not prevent escalation of other nodes, while the second is an optimization that lets us bypass negative laxity nodes since they are not expected to meet their deadlines even without the promotion.

Algorithm 2: is_feasible

```
Function is_feasible(ready_queue, fnode, index):
    can_forward = True;
    for node ∈ ready_queue do
        if ready_queue.index(node) == index then
            break;
        curr_laxity = node.laxity - curTick();
        if not node.is_fwd and curr_laxity > 0 then
            can_forward = curr_laxity > fnode.runtime;
            break;
    if can_forward then
        for node ∈ ready_queue do
            if ready_queue.index(node) == index then
                break;
            node.laxity -= fnode.runtime;
    return can_forward
```

3.2.2 Execution time prediction

Since RELIEF and its feasibility check are laxity-driven, they require an estimate of each node's execution time. We accomplish that by predicting the compute time and memory access time of each task separately.

Compute time prediction: The compute time of fixed-function accelerators, such as the ones used in this study, is largely a function of the input size and the requested computation, owing to the data-independent nature of their control flow [53]. The compute time of such devices can, therefore, be profiled just once at either design time or system boot-up since there will be very little variation. Our evaluation shows that this scheme has an average error of just 0.03% (Section 3.4.6).

Memory time prediction: The memory access time prediction works by predicting two values: the available bandwidth and the amount of data movement. For the former, we experiment with three different predictors based on prior work [69]: *Last value*, *Average*, which computes the arithmetic mean of the bandwidth of *n* previous tasks, and *Exponentially Weighted Moving Average (EWMA)*, that computes a weighted sum of the most recently achieved bandwidth (*bw*) and historical data, as shown below:

$$pred_n = \alpha \times bw + (1 - \alpha) \times pred_{n-1} \tag{3.3}$$

The data movement predictor works by analyzing the graph and observing node states. For predicting input data movement, we need to predict if a node can be colocated with its parent, since colocations eliminate producer/consumer data movement. Given that the scheduler performs colocations by tracking the previously executed node on an accelerator, only one child can be colocated. We predict that the child with the earliest deadline of a set of newly ready children will colocate with the parent if they use the same accelerator type.

For predicting output data movement, we need to predict the number of forwards. If all children can forward from the node, then we will not need to write results back to the main memory. This will be true if a) all the children map to a unique accelerator, and b) all the children will be ready when the node finishes. The former is a simple comparison between the number of tasks mapping to an accelerator type and the instances of that type, while the latter is achieved by ensuring that the node is the latest finishing parent based on its deadline.

The accuracy and performance of bandwidth predictors compared to a *Max* prediction scheme, where the maximum available bandwidth is used, are presented in Section 3.4.6. We also compare the data movement predictor to a *Max* prediction scheme where maximum data movement is assumed.

3.2.3 System architecture

We present the system architecture that we assume in Figure 3.3. The accelerators are modeled to directly access physical memory without address translation, like some existing designs [200]. We propose exposing the entire scratchpad memory in each accelerator to the rest of the system via a non-coherent read-only port. The newly exposed scratchpad memories are not mapped to user address space and access is hidden behind device drivers, ensuring secure access. We also use a discrete hardware manager that is coherent with the CPUs (Section 3.1.2), responsible for scheduling nodes onto accelerators as well as for orchestration of data movement between producers and consumers.

The CPUs, the hardware manager, and the accelerators communicate via shared main memory and interrupts. The CPU informs the hardware manager of new DAGs by writing the root nodes into shared queues in the main memory. Each node is a structure that represents a task for an accelerator, as shown in Table 3.3. The hardware manager parses each node to push them onto ready queues, and launches them on the accelerators via driver functions. The accelerators inform the manager of the completion of each task by raising an interrupt. When a node completes, the manager updates its `status`

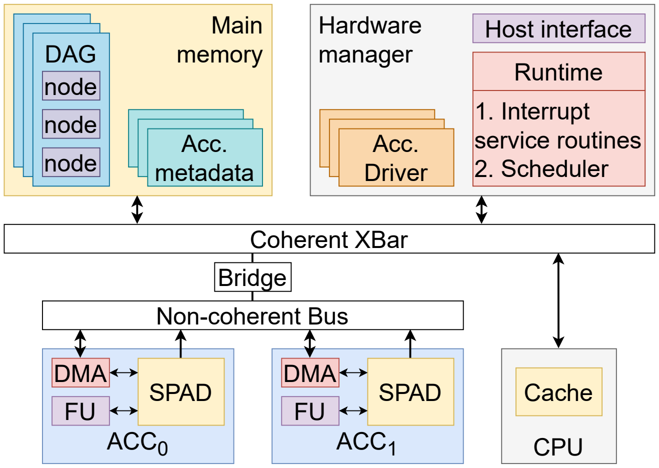


Figure 3.3: System architecture depicting the hardware manager and the interconnect.

field to inform the host CPU program of its completion and pushes its children onto ready queues if their dependencies are satisfied. The user program can learn of the completion of an entire DAG by reading the `status` of leaf nodes.

The node structure contains a few more synchronization and bookkeeping fields that we hide for brevity. The size of the structure depends on the number of parents and children each node has, along with the pointer size. Assuming 32-bit pointers, the base size of the structure with a single parent and child is 72 bytes, with each additional parent and child adding 12 bytes and 4 bytes, respectively. The largest node we see in our applications is 96 bytes. While we show the arrays to be of a constant size, this implementation choice may be replaced with dynamic structures.

Forwarding mechanism

Exposing accelerator private scratchpad memories onto the system interconnect allows consumer DMA engines to perform reads from producer scratchpads without having to go to the main memory. Such a modification should be fairly straightforward in modern SoCs [246], exposing the scratchpad memories to the system interconnect on the DMA plane. This is what we assume in our evaluation. It also possible to leverage PCIe

Table 3.3: DAG node data structure

struct node
`uint32_t acc_id;`
`void *acc_inputs[NUM_INPUTS];`
`node *children[NUM_CHILDREN];`
`node *parents[NUM_INPUTS];`
`uint8_t status;`
`uint32_t deadline;`
`acc_state *producer_acc[NUM_INPUTS];`
`uint32_t producer_spm[NUM_INPUTS];`
`uint32_t completed_parents;`

resizable-BAR support [8], which enables exposure of multiple gigabytes of private accelerator memory into the CPU address space, and Linux P2PDMA interface [172, 240], which allows for direct DMA transfers between PCIe devices.

Hardware manager

We now detail the data structures maintained and runtime executed by the hardware manager described in Section 3.1.2. We chose a microcontroller-based implementation for our work since it offers sufficient performance (Section 3.4.7).

Manager data structures: The hardware manager maintains metadata for each accelerator to track its state and to manage synchronization of data between producers and consumers. Table 3.4 presents the key metadata fields. In addition to maintaining the address for accelerator and DMA engine MMRs (`acc_mmr` and `dma_mmr`), the metadata also holds the address of the scratchpad memory partitions (`spm_addr`), the state of the accelerator (`status`, e.g., idle or running), and the number of accelerators currently reading from each of its scratchpad partitions (`ongoing_reads`). Scratchpad partitions are used to implement multi-buffering.

Table 3.4: Accelerator metadata

struct acc_state
`uint8_t *acc_mmr;`
`uint8_t *dma_mmr;`
`uint8_t *spm_addr[NUM_SPM_PARTITIONS];`
`uint8_t status;`
`node *output[NUM_SPM_PARTITIONS];`
`uint32_t ongoing_reads[NUM_SPM_PARTITIONS];`

The scratchpad partition addresses are physical addresses used by consumer DMA engines to perform direct data transfers. The field `ongoing_reads` is used to keep track of how many consumers are reading from a scratchpad partition of the accelerator to avoid overwriting the data. The manager increments the count before a consumer starts transferring the data to its local scratchpad memory and reduces the count after it is done, thus ensuring that write-after-read dependencies are respected when data is being forwarded.

The metadata size for each accelerator in our implementation, assuming 32-bit pointers and a maximum of 3 scratchpad partitions (`NUM_SPM_PARTITIONS`), is 32 bytes, totaling to 236 bytes for the 7 accelerators our system simulates.

Manager runtime: Alongside launching tasks onto accelerators, the manager runtime implements an interrupt service routine (ISR) and the scheduler. The ISR is triggered every time an accelerator finishes a *job*, where a job could be a DMA operation or computation.

Once an accelerator finishes execution and the scheduler is run, the field `output[p]` (Table 3.4) is set to point to the node that just finished, denoting that partition *p* holds the node's output. The `producer_acc` and `producer_spm` fields are also set in the child nodes to inform their drivers of which producer accelerator and partition to read from. When child nodes are launched, their driver checks if the data is still present in the producer's scratchpad and forwards it if it is. In addition, if all the child nodes are not at the head of their respective ready queue (i.e., not next in line for execution), or the parent node does not have any children, the runtime calls the producer driver to write the results back to main memory immediately.

3.3 Evaluation Methodology

3.3.1 Benchmarks

We evaluate RELIEF against the four policies summarized in Section 3.1 using three vision and two RNN applications. The five applications, along with their input size, deadline, and laxity (when run alone), are listed in Table 3.5. We assume the vision applications run at 60 frames per second (FPS) and thus use a deadline of 16.6 ms. Deadline for RNN applications has been borrowed from previous work [299]. Input sizes mirror prior work as well [57, 299]. Richardson-Lucy deblur is an iterative algorithm where higher iterations lead to better picture quality. We use 5 iterations to have a representative input size balanced with simulation time. Along similar lines, we assume a sequence length of 8 for both LSTM and GRU.

Table 3.5: Vision and machine learning benchmarks

(Symbol) Benchmark	**Input / hidden layer size**	**Deadline**	**Laxity**
(C) Canny edge detection [38]	128 x 128	16.6 ms	13.6 ms
(D) RL deblur [178, 234]	128 x128	16.6 ms	0.2 ms
(G) GRU [49]	128	7 ms	2.3 ms
(H) Harris corner detection [105]	128 x 128	16.6 ms	14 ms
(L) LSTM [114]	128	7 ms	3.6 ms

3.3.2 Platform

We use gem5-SALAM [236] for our evaluation, which provides a cycle-accurate model for accelerators described in high-level C. The simulator consumes the description of an accelerator in LLVM [177] intermediate representation (IR) and a configuration file and provides statistics like execution time and energy consumption. These accelerators are then mapped into the simulated platform's physical address space, enabling access via memory-mapped registers. The simulated configuration, listed in Table 3.6, models a typical mobile device [196]. We model the hardware manager using an ARM Cortex-A7 based microcontroller running bare-metal C code. Cortex-A7 has an area and power overhead of 0.45mm^2 and $<$100mW [23], which can be reduced further by stripping the vector unit. The simulated platform models end-to-end execution of applications, from inserting the tasks into ready queues till the completion of each requested application. This includes interrupt handling, scheduling, driver functionality, DMA transfers, and accelerator execution. In addition to the bus-based interconnect between the accelerators listed in Table 3.6, we evaluate RELIEF's performance with a crossbar switch in Section 3.4.8. The two topologies represent two ends of the interconnect cost/performance spectrum.

Table 3.6: Simulation setup

Hardware manager	ARM Cortex-A7 based 1.6 GHz single-core in-order CPU 32 KB 2-way L1-I + 32 KB 4-way L1-D 64 B cache line size
Main memory	LPDDR5-6400; 1 16-bit channel; 1 rank; BG mode t_{CK} = 1.25ns; burst length = 32 Peak bandwidth = 12.8 GB/s
Interconnect	Full-duplex bus; width = 16 B Peak bandwidth = 14.9 GB/s

Our evaluation uses seven image processing accelerators, one each for the kernels shown in Figure 3.1. Each accelerator was designed in isolation by determining the *energy*×*delay*^{2} *(ED*^{2}*)* product for the execution of a single task on the accelerator, while varying the configuration in terms of the number of functional units and memory ports. The configuration with the minimum *ED*^{2} was chosen for the design, similar to previous work [236, 251]. In practice, we expect accelerators to work on the same input size to allow for easy chaining and sharing of data by commonly used applications. Our accelerators, clocked at 1 GHz, thus, have enough scratchpad memory to work on 128x128 inputs along with double buffered output to avoid blocking on consumer accelerator reads. The precise scratchpad memory sizes are listed in Table 3.1. For accelerators with differing input sizes, the software runtime or the hardware manager can break down tasks into smaller chunks, similar to accelerator composition in GAM+ [57].

3.3.3 System load

Combinations of the applications in Table 3.5 are often seen in real-world scenarios, e.g., Canny+LSTM is used for lane detection in self driving cars [296]. Enumerating all combinations of these applications, thus, helps us cover all their existing and potential future use cases. We experiment with four levels of contention to see how each of the policies scale. *Low* contention is just a single application, *medium* contention is all combinations of size 2, while *high* contention is all combinations of size 3. Increasing contention represents reduced ability to meet deadlines, with combinations larger than 3 meeting very few deadlines and thus not evaluated. In each of these scenarios, each application is instantiated once and the simulation ends when the last application finishes execution. The fourth level of contention, called *continuous* contention, is a modification of *high* contention where each of the three applications are run in a continuous loop to ensure each application experiences contention throughout its execution. We limit the execution time of each simulation to 50ms and report results for finished tasks. Each application is represented with a symbol in the following figures, as listed in Table 3.5.

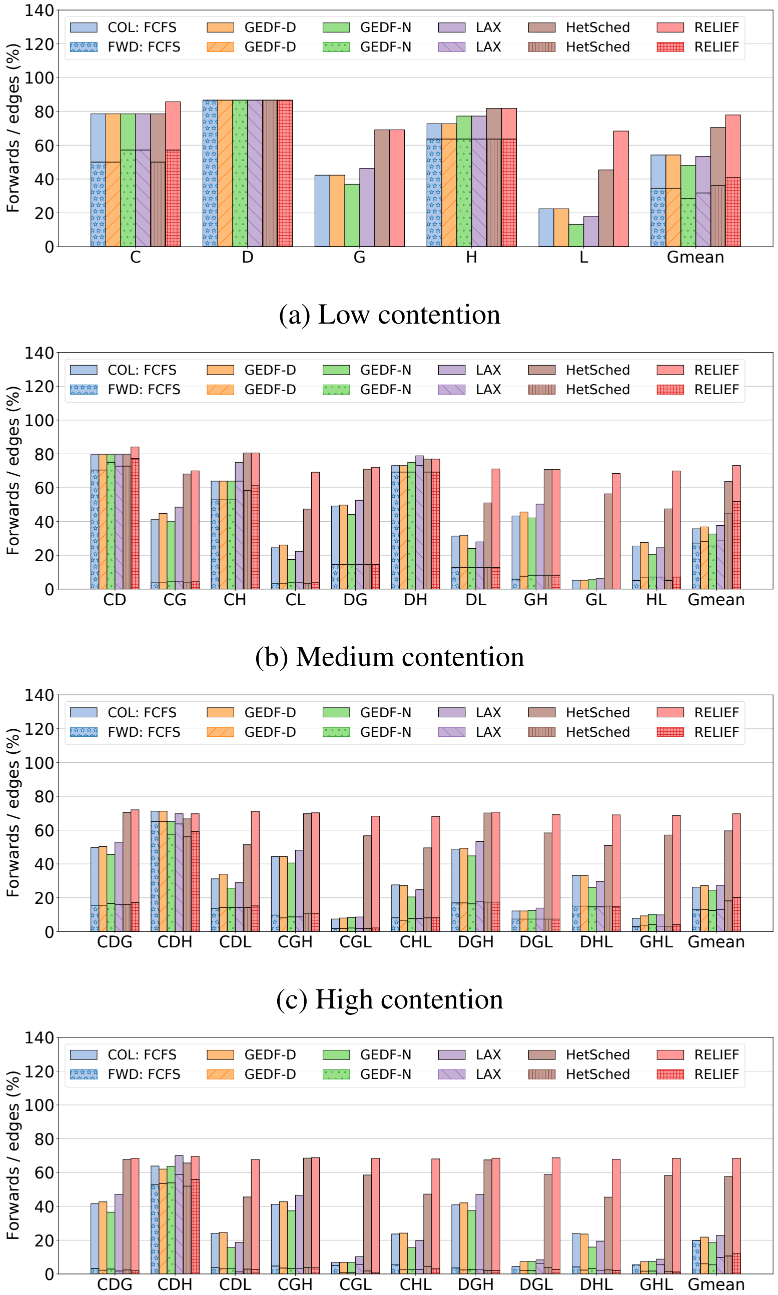
3.4 Results

3.4.1 Data forwards

Our primary design goal with RELIEF is producing more data forwards than SOTA policies. We quantify this increase in Figure 3.4.

Observation 1: SOTA policies under-utilize forwarding mechanisms. In contrast, RELIEF consistently achieves >65% of all possible forwards, on average. This is clear from Figure 3.4, which shows the percentage of total data forwards and colocations, computed as the ratio of number of forwards/colocations to the total number of edges in the mix. We can see how SOTA policies' obliviousness to data forwarding mechanisms leads to their under-utilization, achieving as little as 8% of all forwards possible. In contrast, RELIEF improves over HetSched, the leading SOTA policy, by nearly 1.2x on average under continuous contention.

We observe two trends across all three four of contention in Figure 3.4: 1) RNN applications (GRU and LSTM) are the biggest contributors to colocations, and 2) application mixes with more RNN applications achieve better forwarding with RELIEF than others. The first observation is unsurprising given that all RNN tasks map onto a single resource. For the second observation, we attribute the gains with RNN applications to the fact that they contain long, linear chains (up to 9 nodes) that have the same structure and node deadlines. Having the same node deadlines means that deadline-aware policies schedule each of those chains in a round-robin fashion, thus forfeiting any forwarding opportunities. FCFS has a similar problem of being locality oblivious. HetSched is able to achieve significantly more forwards than other baseline policies. These gains stem primarily from HetSched's ability to prioritize GRU's critical path, which happens to contain most of its forwards.



(a) Low contention

(b) Medium contention

(c) High contention

(d) Continuous contention

Figure 3.4: Percent of total forwards and colocations, computed as the ratio of the total number of forwards/colocations to the total number of edges in the mix.

3.4.2 Data movement

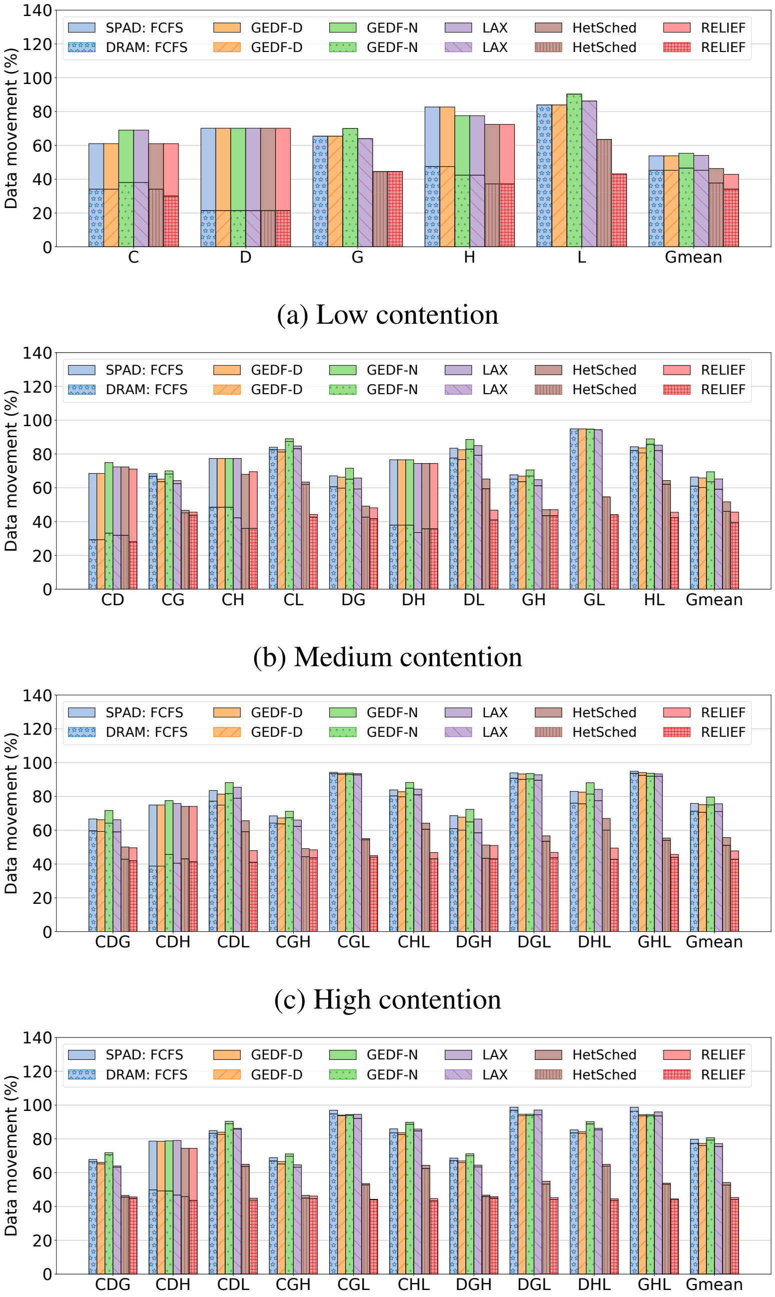
To understand each policy's data movement behavior, Figure 3.5 plots the percentage of data transfers (in bytes) that materialize as main memory accesses, scratchpad-to-scratchpad transfers, and colocations.

Observation 2: RELIEF reduces main memory traffic by up to 32% compared to HetSched, under each level of contention. The average reduction compared to HetSched rests at 10%, 14%, 16%, and 16%, for low, medium, high, and continuous contention, respectively. This is a key result and highlights how simple changes to the scheduler can yield significant reductions in memory traffic.

The percentage of forwards that materialize as colocations in a mix is a function of its application composition. As explained before and evident from Figure 3.5a, all GRU and LSTM forwards are colocations since these applications map to a single accelerator. In contrast, the vision applications are more diverse in their resource needs and exhibit a greater degree of scratchpad-to-scratchpad data movement. The behavior of single applications impacts the behavior of entire mixes. Mixes CD, CH, and DH (medium contention), for instance, have fewer colocations than other mixes. The same is true for mix CDH (high/continuous contention).

The reduction in data movement traffic reduces energy consumption for both the main memory *and* scratchpad memories. We quantify this reduction for the high contention scenario in Figure 3.6.

Observation 3: RELIEF reduces main memory and scratchpad memory energy consumption by up to 18% and 8%, respectively, compared to HetSched under high contention. The average main memory and scratchpad energy reduction compared to HetSched is 7% and 4%, respectively. Forwards reduce main memory traffic while colocations eliminate both main memory and scratchpad memory traffic. While forwards cause an increase in scratchpad activity, colocations more than make up for the increase. RELIEF has the same scratchpad energy consumption as LAX for



(a) Low contention

(b) Medium contention

(c) High contention

(d) Continuous contention

Figure 3.5: Breakdown of data movement into main memory traffic (lower bars), SPAD-to-SPAD traffic (upper bars), and colocations (empty space). Data is normalized to total data movement when all loads and stores go to main memory.

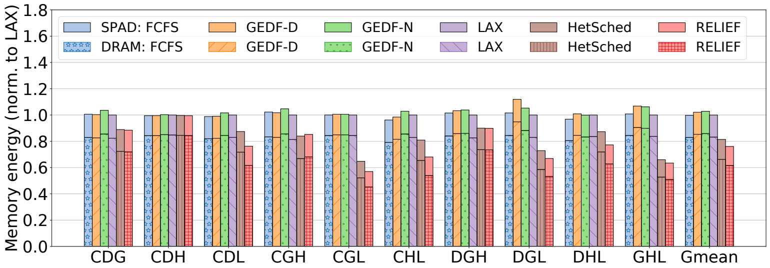


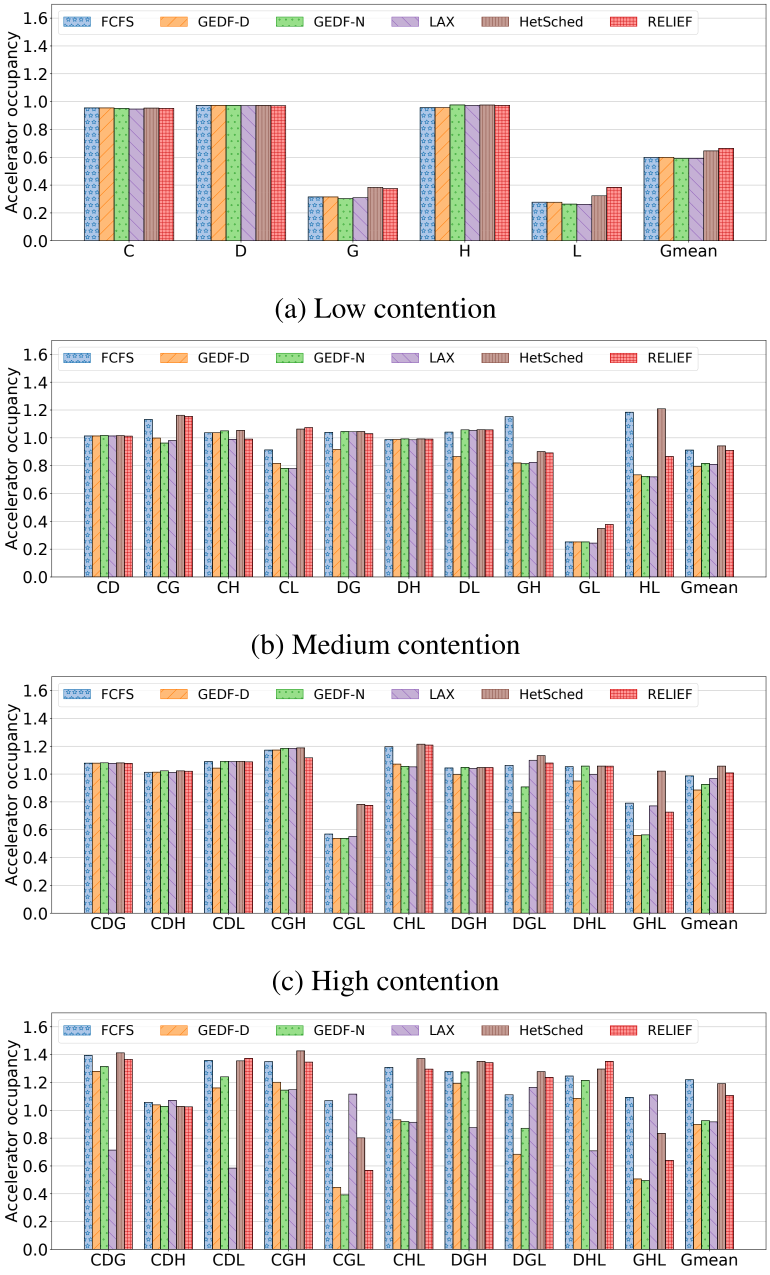
Figure 3.6: Total main memory and scratchpad memories' energy consumption under high contention using gem5-SALAM's energy models.

CDH, for instance, but reduces it by 24% for CGL.

3.4.3 Accelerator utilization

Figure 3.7 shows accelerator utilization (or occupancy), defined as the sum, across all accelerators, of the fraction of total execution time for which each accelerator was busy. Accelerator occupancy provides a measure of degree of parallelism in each scenario. Note that while the numerator is relatively constant under the low, medium, and high contention scenarios, the denominator, which is total execution time, is impacted both by the degree of computational parallelism and by the data movement cost resulting from the use of each policy. For the continuous contention scenario, the denominator remains constant, while the numerator is impacted by the number and type of nodes executed, the data movement cost, and the degree of computational parallelism, all of which vary by policy.

Observation 4: RELIEF improves accelerator utilization by up to 41%, compared to LAX under high contention, with an average improvement of 4%. HetSched, in turn, results in best case and average improvements of 41% and 5% relative to RELIEF, respectively. RELIEF's improvements over LAX are a result of increased number of forwards, resulting in lower execution time. In its attempt to increase forwards, RELIEF can sometimes hinder the progress of tasks whose chil-



(a) Low contention

(b) Medium contention

(c) High contention

(d) Continuous contention

Figure 3.7: Accelerator occupancy is defined as ratio of the sum of total of all accelerators' compute time to the the end-to-end system execution time, measured from the initiation of all applications to the completion of the last application. Higher is better.

dren map to different accelerators, resulting in a lower degree of parallelism. This is especially evident in mixes CGL and GHL under continuous contention, where GRU and LSTM tasks, all of which map to `elem-matrix`, get promoted frequently, limiting the time they execute in parallel with the vision tasks, which utilize a variety of accelerators. HetSched and LAX's gains over RELIEF for these application mixes are primarily attributed to RELIEF's lower accelerator-level parallelism and increased scheduling latency (Section 3.4.7).

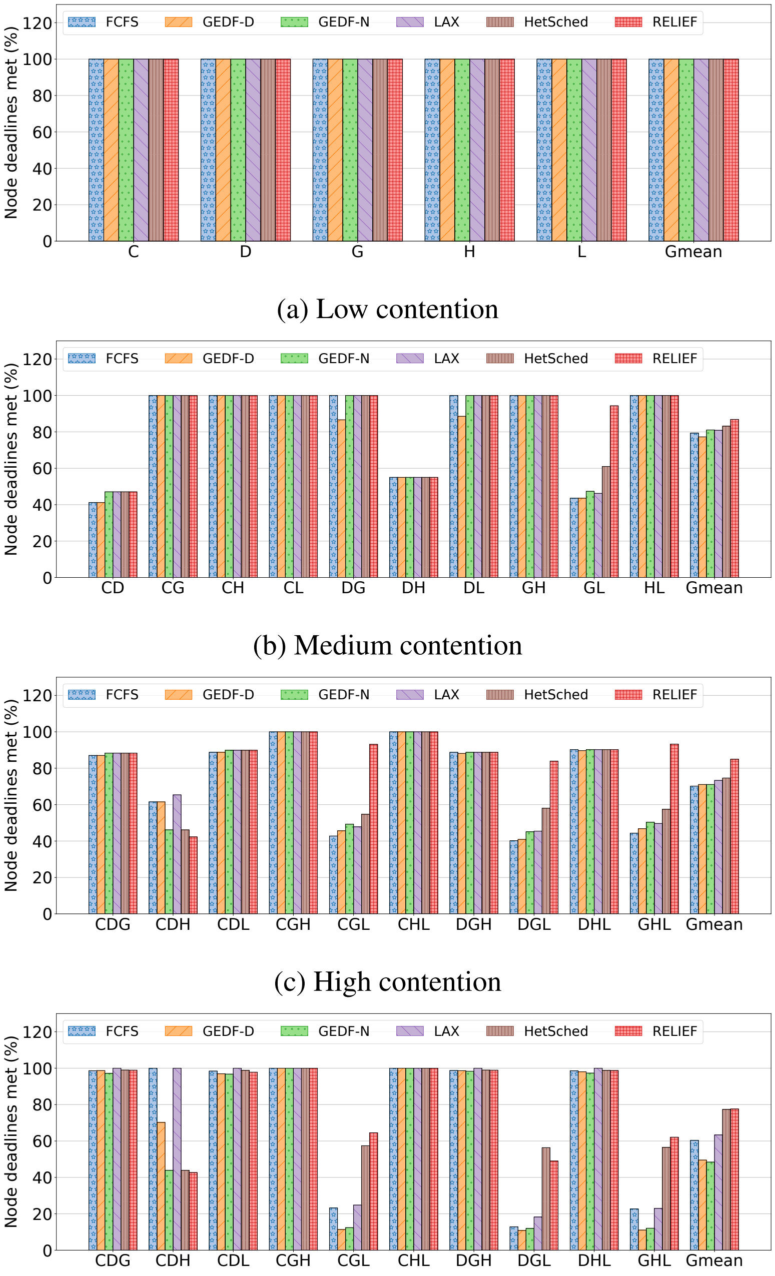
While RELIEF's promotions reduce the degree of parallelism on average relative to HetSched, they do not cause unfairness. In fact, it is a fairer policy when compared to LAX and HetSched, as we will see in Section 3.4.5.

3.4.4 Node deadlines met

RELIEF integrates a feasibility check (Section 3.2) that makes a best-effort to minimize missed deadlines. To evaluate its efficacy, we compute the percentage of node deadlines met in each application mix and present the results in Figure 3.8.

Observation 5: RELIEF meets up to 70% more node deadlines compared to HetSched, under high contention, with an average improvement of 14%. More importantly, RELIEF rarely *reduces* the number of deadlines met compared to SOTA. This highlights the effectiveness of the feasibility check in throttling priority elevations to prevent deadline violations.

The only instance where RELIEF performs worse than existing policies is in the high contention mix CDH. We observe that GEDF-N and RELIEF prioritize Deblur nodes over Canny and Harris nodes since the former have a lower deadline and laxity. This causes nearly all of the Canny and Harris nodes to miss their deadlines. Furthermore, not all Deblur nodes meet their deadlines either because of high contention. HetSched has a similar story of prioritizing Deblur due to its longer critical path. LAX's ability to de-prioritize applications with negative laxity allows it to de-prioritize Deblur,



(a) Low contention

(b) Medium contention

(c) High contention

(d) Continuous contention

Figure 3.8: Percent of node deadlines met.

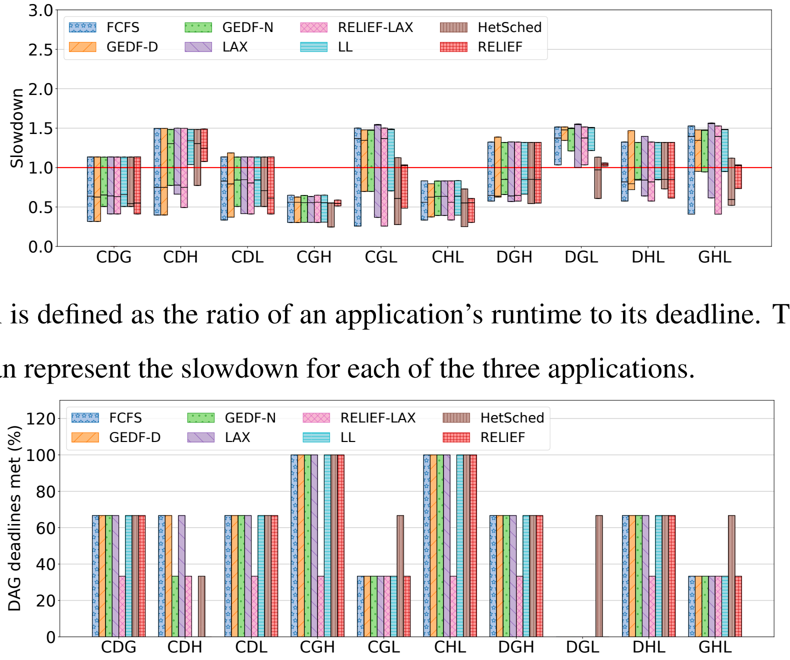
allowing all Canny and Harris nodes to make progress. FCFS does not suffer from this problem either because it does not prioritize DAGs and nodes. GEDF-D has the same schedule as FCFS given that all the DAGs in this mix have the same deadline. RELIEF also performs worse than HetSched in DGL, but the latter achieves the gains by unfairly slowing down LSTM. We will explore fairness in more detail in Section 3.4.5.

Continuous contention has a different setup compared to the other three scenarios, as described in Section 3.3.3. Under continuous contention, each mix executes a different number and type of nodes under different policies for a fixed period of time. In the other three scenarios, each application in a given application mix runs to completion and executes exactly once, so the number of nodes executed is constant across policies with the execution time depending on the policy's scheduling decisions. This different simulation setup results in what looks like anomalous behavior of a higher percentage of deadlines met under continuous contention compared to high contention (e.g., CDG), but in reality they cannot be directly compared. This hints at a tradeoff between deadlines met and fairness that we explore in the next section.

3.4.5 Quality-of-Service and Fairness

An important aspect of RELIEF's design is fairness: increased forwards for one application should not come at the cost of excessive slowdown for others. Figure 3.9a shows a box plot of application slowdown in each mix under high contention. The figure also shows the results for LL and *RELIEF-LAX*, a variant of RELIEF that integrates LAX's de-prioritization mechanism (Section 3.1.3). Figure 3.9b, meanwhile, plots the percent of DAG deadlines met under high contention.

Figure 3.9a shows how RELIEF reduces maximum slowdown and variance by up to 17% and 93%, respectively, compared to HetSched. The latter meets the same or more DAG deadlines across the board, however (Figure 3.9b). The two results highlight a key tradeoff: HetSched meets more DAG deadlines by unfairly slowing down one



(a) Slowdown is defined as the ratio of an application's runtime to its deadline. The box edges and the median represent the slowdown for each of the three applications.

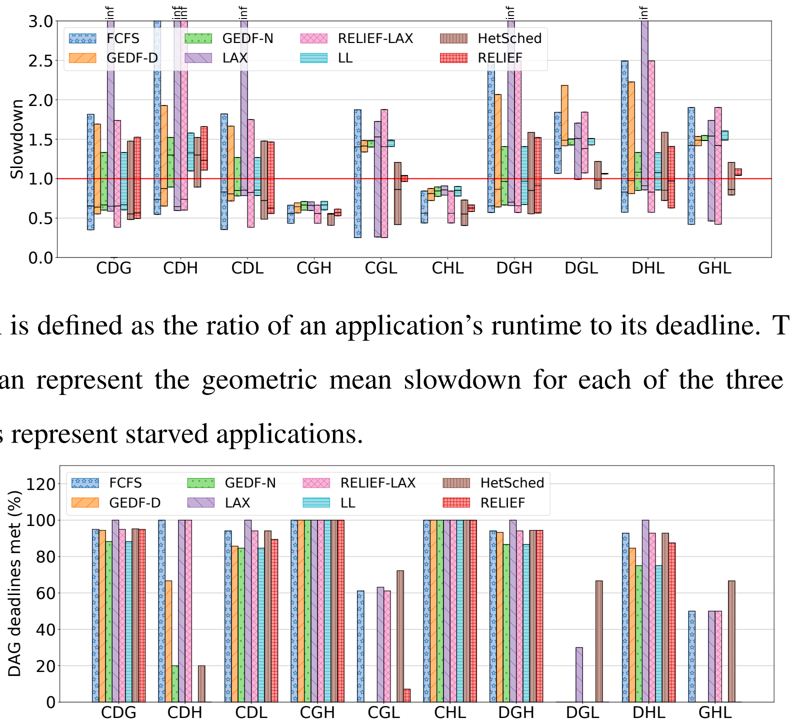
(b) Percent of DAG deadlines met.

Figure 3.9: Slowdown (a) and DAG deadlines met (b) under high contention.

application over another, as evident from its wider slowdown spread, while RELIEF attempts to distribute slowdowns and allows each DAG to make progress commensurate with its deadline. This tradeoff is made even more evident under continuous contention, as shown in Figures 3.10a and 3.10b.

Observation 6: RELIEF improves fairness, reducing worst-case deadline violation and variance by up to 14% and 98%, respectively, compared to HetSched under continuous contention. HetSched is able to meet more DAG deadlines (Figure 3.10b) and improve accelerator utilization (Section 3.4.3) by unfairly favoring some applications over others. For instance, HetSched meets 10 DAG deadlines in DGL while RELIEF meets 0, but it does so by slowing down one application (LSTM) by 22%. In contrast, every application suffers a slowdown of <7% under RELIEF , accompanied by a 98% reduction in variance.

We also see how LAX's de-prioritization mechanism causes significant unfairness



(a) Slowdown is defined as the ratio of an application's runtime to its deadline. The box edges and the median represent the geometric mean slowdown for each of the three applications. Infinite values represent starved applications.

(b) Percent of DAG deadlines met.

Figure 3.10: Slowdown (a) and DAG deadlines met (b) under continuous contention.

in mixes CGL, DGL, and GHL. In all three cases, the RNN applications start missing deadlines early on due to contention and are de-prioritized by LAX and RELIEF-LAX in favor of the vision applications, causing significant unfairness. This is especially troublesome considering that they have lower deadlines compared to vision applications (Table 3.5). In contrast, RELIEF allows the RNN applications to progress alongside the vision applications, ensuring more deadlines are met while reducing unfairness.

LAX also has a starvation problem, as is made evident from Figure 3.10a and Table 3.7. The table lists the number of completed DAG iterations for each application in each continuous contention mix. We see how Deblur is starved in every mix it is in except DGL. Deblur is extremely sensitive to queuing delays given its laxity of just 0.2ms (Table 3.5). Combined with its linear task graph, this means that if even a single Deblur node is delayed by more than 0.2ms, the node's laxity will drop below 0 and it will get

Table 3.7: Number of finished DAGs in each application mix under continuous contention.

Policy	C	D	G	C	D	H	C	D	L	C	G	H	C	G	L	C	H	L	D	G	H	D	G	L	D	H	L	G	H	L
FCFS	8	1	11	4	0	4	8	1	8	5	11	5	11	3	4	5	5	8	1	11	5	2	3	4	1	5	8	3	7	4
GEDF-D	5	1	12	3	1	2	3	2	9	5	11	4	2	4	4	3	3	9	1	11	3	1	4	4	1	3	9	4	2	4
GEDF-N	4	2	11	2	1	2	3	2	8	4	11	4	2	4	4	3	3	8	1	11	3	1	4	4	1	3	8	4	2	4
LAX	5	0	11	5	0	5	3	0	8	4	11	4	12	3	4	3	3	8	0	11	4	3	3	4	0	3	8	3	7	4
RELIEF-LAX	8	1	11	4	0	4	8	1	8	5	11	5	11	3	4	5	5	8	1	11	5	2	3	4	1	5	8	3	7	4
LL	4	2	11	2	1	2	3	2	8	4	11	4	2	4	4	3	3	8	1	11	3	1	4	4	1	3	8	4	1	4
HetSched	6	1	14	2	1	2	6	1	10	6	14	5	6	7	5	6	5	10	1	14	3	3	7	5	1	3	10	7	3	5
RELIEF	5	1	14	2	1	2	5	2	12	5	14	5	2	6	6	5	4	12	1	14	3	2	6	6	1	3	12	6	2	6

deprioritized by LAX. This is precisely what happens when Deblur contends with other vision applications for the `convolution` accelerator: if any node is launched on the `convolution` accelerator while a Deblur node is waiting, the latter will be stalled for at least 1.5ms (Table 3.2), causing starvation. This stalls any progress for Deblur until the system has no other node to offload to the `convolution` accelerator. DGL does not suffer from this problem because GRU and LSTM do not use the `convolution` accelerator. FCFS also has 0 finished Deblur iterations in CDH, but our experiments show that it is not starved; rather it is making slow progress.

3.4.6 Prediction accuracy

The feasibility check presented in Section 3.2 utilizes a predictor to estimate compute and memory access times for accelerators. Table 3.8 presents the error in the compute time, the data movement, and the different memory bandwidth predictors under high contention, along with the latter's impact on the number of forwards and node deadlines met. We empirically chose *n=15* for *Average* and $\alpha = 0.25$ for *EWMA* for the best accuracy.

Observation 7: Compute time prediction has a maximum error of just 0.03%. This validates prior observations that compute time can be defined as a function of input size and requested operation for fixed function accelerators [53].

Table 3.8: Accuracy of compute time and data movement predictors, along with the accuracy and performance of memory bandwidth predictors. Negative error values represent underestimation of true value while positive error values represent overestimation. The geometric mean uses absolute error values.

Mix	Compute	Memory DM	Memory BW error (%)				Forwards				Node deadlines met			
	error (%)	error (%)	Max	Last	Average	EWMA	Max	Last	Average	EWMA	Max	Last	Average	EWMA
CDG	0.06	-0.95	-56.33	5.85	-1.24	1.1	139	138	138	139	136	136	136	136
CDH	0	-8.06	-59.03	-19.42	-3.95	-4.68	46	46	47	47	22	22	22	22
CDL	-0.05	-0.88	-56.47	5.19	-1.27	2.02	155	155	155	155	160	160	160	160
CGH	0.1	-1.01	-55.7	7.13	-1.18	2.19	130	130	130	130	150	150	150	150
CGL	0.02	0.59	-55.39	11.23	0.42	4.37	230	230	232	231	257	255	254	252
CHL	0.05	-0.93	-56.63	5.93	-0.64	2.79	143	143	143	143	174	174	174	174
DGH	0.03	-3.14	-56.94	4.26	-1.33	0.96	142	142	142	142	142	142	142	142
DGL	-0.02	-2.15	-55.5	8.95	-0.07	2.67	244	245	244	245	240	242	239	242
DHL	0	-3.33	-56.7	3.65	-1.36	1.31	156	156	157	157	166	166	166	166
GHL	-0.05	-0.57	-55.41	11.13	0.09	3.06	237	238	239	238	263	261	260	258
Gmean	0.03	1.47	56.4	7.31	0.68	2.22	-	-	-	-	-	-	-	-

Data movement prediction also works well, with an average error of 1.35%. Memory bandwidth predictors, meanwhile, exhibit a range of accuracies, with *Average* performing the best both in terms of mean (0.68%) and maximum (3.95%) error. Their accuracy has little to no impact on performance, however. We can see from Table 3.8 how each policy achieves essentially the same number of forwards and deadlines met.

To understand the incremental impact of data movement and memory bandwidth predictors, Figure 3.11 plots the performance impact of the two predictors in isolation and combined, normalized to having *Max* predictor for both. The bandwidth predictor here is *Average*. We can see how little impact the accuracy of the predictor has on RELIEF's ability to meet deadlines. Their impact on forwards (not shown) is similar.

Observation 8: RELIEF does not benefit from dynamic memory time prediction. Each application has several forwarding chains, which are contiguous sequence of forwarding producers/consumers. The laxity calculation based on the memory time prediction decides how these chains get broken up into sub-chains and interleaved. We notice that the number of sub-chains produced by each predictor does not differ

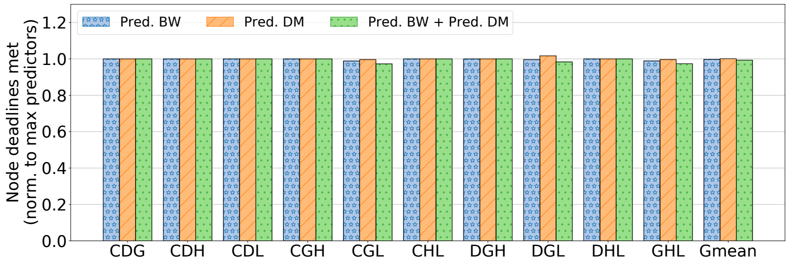


Figure 3.11: Impact of memory predictors on missed deadlines under high contention.

significantly, which is why they all achieve similar overall performance. Given this observation, we have used the baseline *Max* predictors for all our evaluations since they offer the same performance for negligible overhead.

3.4.7 Scheduler execution time

The execution time of a scheduling policy is an important factor in choosing one, since a better schedule may not offset the overhead of preparing the schedule itself. Figure 3.12 plots the average and tail latency of pushing a task into the ready queue for each policy on a Cortex-A7 based microcontroller.

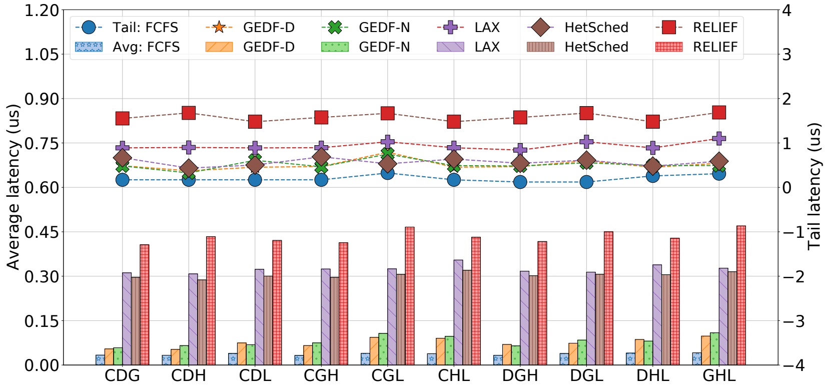


Figure 3.12: Average (bars) and tail (lines) latency of the scheduler with different policies on a Cortex-A7 based microcontroller, under high contention.

Observation 9: RELIEF has higher overhead than existing policies, but is easily overlapped with accelerator execution. Looking at Figure 3.12 and Table 3.2, we can see that RELIEF's modest scheduling overhead can be easily overlapped with computation, minimizing its contribution to the critical path.

3.4.8 Impact of interconnect topology

A crossbar is a high-throughput switch allowing up to $n \times m$ concurrent transactions for n requesters and m responders. This should benefit RELIEF since it permits concurrent transactions between independent producer/consumer pairs. Figure 3.13 shows RELIEF's sensitivity to the interconnect in terms of interconnect occupancy and the total execution time, under high contention.

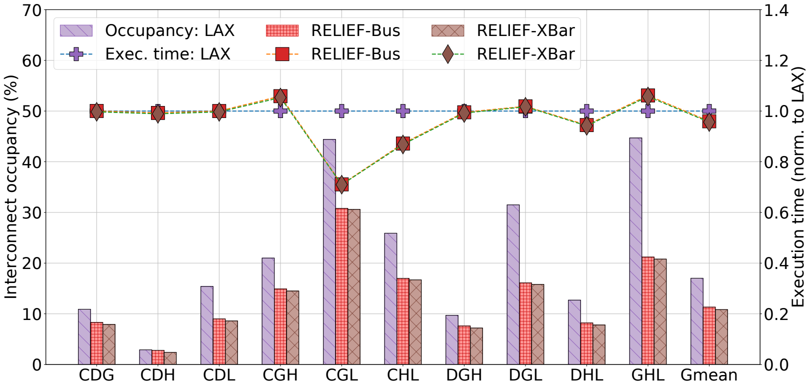


Figure 3.13: RELIEF's sensitivity to system interconnect under high contention. Interconnect occupancy is defined as the percentage of cycles for which the interconnect had at least one transaction going through.

Observation 10: RELIEF reduces interconnect occupancy by up to 49% compared to LAX, with an average reduction of 33%. It does not, however, benefit from high-performance interconnects. RELIEF's low interconnect occupancy is a result of its reduction in data movement (Section 3.4.2) as well as a lack of accelerator-level parallelism (Section 3.4.3). This indicates that these applications are not interconnect-bound, an observation further supported by the fact that the average queuing delay for the bus is less than a cycle (not shown). We expect applications with more varied resource needs and larger input sizes to benefit more from complex interconnects.

3.5 Related Work

The general scheduling problem of running *n* tasks on *m* processors such that the run time is minimized has been proven to be NP-complete [269]. Nevertheless, several heuristics have been proposed along with optimal solutions for more constrained versions of the problem. This subsection discusses some of these solutions, both for traditional CPU/GPU systems as well as more heterogeneous and real-time systems.

GPU scheduling: Prior work in GPU scheduling has looked at co-scheduling and distributing work across CPUs and GPUs to reduce synchronization and data-movement overhead [98, 179]. PTask [237] optimizes for fairness and tries to reduce data movement by scheduling child tasks onto the same device as the producer when possible. Cilk [31] also implements a child-first scheduling policy that improves locality but it optimizes primarily for improved hardware utilization. While being child-first, both PTask and Cilk are deadline blind, rendering them unsuitable for real-time applications. Zahaf et al. [303] use an EDF policy to determine which device each node should be mapped to (e.g., GPU, DSP) such that all DAG deadlines are met. Their work can be extended by optimizing for better colocation using RELIEF.

Baymax [41] and Prophet [42] use online statistical and machine learning approaches, respectively, to predict whether an accelerator can be shared by user-facing applications and throughput-oriented applications at the same time, without violating the former's QoS requirements. RELIEF can be extended with Baymax and Prophet to efficiently utilize multi-tenant accelerators like GPUs.

Menychtas et al. [188] present a fair queuing-based scheme where the OS samples each process' use of accelerators in fixed time quanta and throttles their access to provide fairness.

Accelerator scheduling: Gao et al. [85] batch identical task DAGs across multiple user-facing RNN applications together for simultaneous execution on a GPU, thereby improving GPU utilization and reducing inference latency. PREMA [53] utilizes a token-based scheduling policy for preemptive neural processing units (NPUs) that distributes tokens to each task based on its priority and the slowdown experienced due to contention, balancing fairness with QoS. While both policies are QoS-aware, neither of them optimize for data movement across multiple accelerators.

GAM+ [57] is a hardware manager that decomposes algorithms into accelerator tasks and schedules them onto physical accelerator instances using a preemptive round-robin policy. The hardware manager we used is based on GAM+. VIP [196] is an accelerator virtualization framework that uses a hardware scheduler at each accelerator to arbitrate among different applications' tasks. The authors use an EDF scheme where the FPS of the application serves as the deadline. Yeh et al. [299] propose exposure of performance counters in GPUs that drives LAX, a non-preemptive least laxity-based scheduling policy. HetSched [12] is another laxity-driven scheduling policy for autonomous vehicles that takes task criticality and placement into account. The scheduling policies underlying these systems are used in our comparative evaluation in Section 3.4.

Real-time scheduling: Optimal scheduling using a job-level fixed priority algorithm is provably impossible [116], unless task release times, execution times, and deadlines are known *a priori* [66]. Baruah presented optimal but NP-complete integer-linear programming formulations [28] along with approximate linear-programming relaxations [29] for scheduling real-time tasks on heterogeneous multiprocessors. Previous work also exists on providing tighter bounds on the response time of the system under both preemptive and non-preemptive variants of GEDF [241, 294]. These mathematically sound formulations provide strong performance guarantees but tend to be infeasible in an online setting. RELIEF's goal is to meet application-specified deadlines while minimizing data movement using a fast, online heuristic approach.

3DSF [239] is a hierarchical scheduler for cloud clusters that integrates three schedulers. The top layer avoids missed deadlines by using a least-laxity (LL) scheme to prioritize deadline constraint jobs over regular ones when necessary, the middle layer minimizes data movement by queuing tasks on servers that have the most inputs available locally, and the bottom layer allocates server resources to each running job proportional to its requirements. Although locality aware, 3DSF has multiple optimization targets that come with execution time overheads untenable for micro-second latency tasks.

3.6 Summary and Discussion

RELIEF (RElaxing Least-laxIty to Enable Forwarding) is an online least laxity-based (LL-based) scheduling policy that exploits laxity to improve forwarding hardware utilization by leveraging one application's laxity to reduce data movement in another application. RELIEF increases direct data transfers between producer/consumer accelerators by up to 50% compared to SOTA, lowering main memory traffic and energy consumption by up to 32% and 18%, respectively. Simultaneously, RELIEF improves QoS by meeting 14% more task deadlines on average, and improves fairness by reducing the worst-case deadline violation by 14%. RELIEF integrates into existing architectures with hardware forwarding mechanisms and a hardware manager, requiring minimal software changes.

While we have demonstrated our ideas over LL-based scheduling, the techniques can be applied over other laxity-based policies such as HetSched as well. LL and HetSched differ in the manner in which laxity is distributed across nodes in a DAG, resulting in scheduling differences in the baseline policies. LL does not distribute its laxity, which means that each node has laxity equal to the current DAG laxity. HetSched, meanwhile, attempts to distribute the laxity among nodes based on their contribution to the critical-path execution time. With LL as a baseline policy, RELIEF has all of DAG's laxity at its disposal that it can choose to exploit whenever it sees fit. With HetSched as a baseline, however, the DAG's laxity is distributed across the nodes, limiting the number of promotions a node will allow. We are currently investigating the impact of using HetSched's laxity calculation in RELIEF. Our preliminary results indicate that such a combination continues to offer significant data movement cost savings, potentially increasing both forwards and deadlines met. We observe, however, that the choice of laxity distribution presents a tradeoff between QoS and fairness.

4 Concurrent PIM and Load/Store Servicing in PIM-Enabled Memory

Modern applications, ranging from consumer mobile applications [33] to large server workloads [75, 276], are becoming increasingly memory bound. While newer memory technologies like HBM [131, 139], GDDR6 [132], and HMC [100, 130] reduce the memory bottleneck by offering wider links, increased parallelism, and improved scalability [39], they still struggle to close the gap between processor and memory performance, the so-called “memory wall” [110, 287].

Processing in-memory (PIM) [3, 84, 99, 108, 161, 195, 270] is a paradigm shift in how we design our computers, dictating that compute be moved closer to data instead of the other way around. PIM architectures place compute units close to/inside main memory cells, minimizing data movement costs and achieving wide data parallelism. A common mechanism to trigger computation on these compute units is by submitting PIM requests, requests that resemble regular memory requests (henceforth called MEM requests) but perform computation in-place rather than moving data to/from the host processor. To handle this heterogeneity in request types, the memory controller needs to switch between *MEM mode* and *PIM mode* to service requests of each type. This switching adds a new dimension to memory controller scheduling that is distinct from traditional architectures.

Though PIM-enabled memories can be paired with many host processors, GPUs

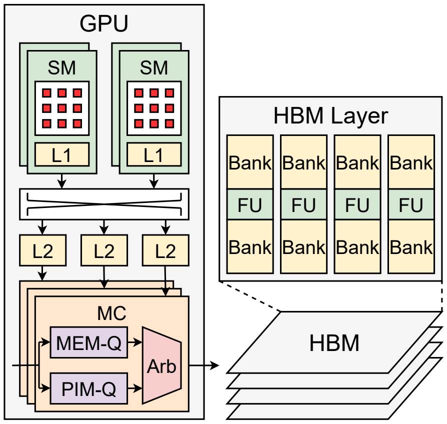


Figure 4.1: PIM-enabled GPU architecture. Each HBM layer contains eight functional units (FUs) that are shared by two banks each (fewer shown in figure for readability).

have emerged as a particularly attractive fit given their highly data-parallel architecture [3, 108, 215]. Figure 4.1 shows an example PIM-enabled GPU architecture. Modern GPUs often utilize techniques like streams [13, 106, 205], multi-process service (MPS) [207], and multi-instance GPU (MIG) [206] to execute multiple kernels concurrently, allowing for improved utilization of the GPU's resources. These techniques enable simultaneous use of both GPU cores and PIM functional units (FUs) by allowing for co-execution of traditional GPU kernels with kernels that submit PIM requests to the main memory for computation [3, 215] (henceforth referred to as GPU and PIM kernels, respectively), improving both resource utilization and application performance. Such simultaneous use can be either *collaborative*, where both GPU and PIM kernels belong to the same application (e.g., large language models [111, 212, 248], graphics [290], and scientific computing [119]), or *competitive*, where two separate applications launch kernels on the two resources [48]. In both cases, the resulting simultaneous use of memory by both GPU and PIM kernels raises an important question: *how do we efficiently route and schedule MEM and PIM requests to ensure fairness between co-executing kernels while also maximizing system throughput?*

To answer this question, we need to look at the two key resources that are shared by

MEM and PIM requests: 1) the memory controller, and 2) the interconnect between the GPU's streaming multiprocessors (SMs) and the memory controllers. We first discuss contention at the **memory controller**. Figure 4.1 shows how the memory controller (MC) maintains separate queues for MEM and PIM requests, with an arbiter (Arb) to switch between them. The switching policy has a direct impact on system efficiency since switching modes: 1) requires the draining of in-flight requests, potentially causing bank idle time, and 2) can hurt locality since MEM and PIM requests often map to different rows. These factors influence queueing delay and the rate at which each request type is served, thereby directly affecting the performance of both GPU and PIM kernels. Optimizing for both system throughput and application fairness is a hard problem, since throughput favors infrequent switching while fairness favors frequent switching. This requires the design of a smart switching policy that can balance the two goals.

PIM kernels are optimized to fully utilize SM resources to send as many PIM requests as possible. Since the memory controller may not be able to keep up with this burst of requests under contention, PIM requests can quickly fill up the memory controller queues and create backpressure in the **interconnect**, causing denial of service to MEM requests. Not only can this stall the SMs executing GPU kernels, but also reduce the memory controller's visibility into the load/store stream and lead to poorer decision making.

Motivated by these challenges, this chapter makes the following contributions:

- A comprehensive analysis and characterization of GPU/PIM co-execution on a PIM-enabled GPU under 180 competitive scenarios and a GPT-3-like collaborative scenario, focused on interconnect and memory controller bottlenecks. We discover that PIM kernels can easily overwhelm the shared memory subsystem by its high request injection rate, causing unfairness. Concurrently, inefficient switching at the memory controller can further exacerbate fairness and throughput bottlenecks.

- Evaluation of the efficacy of adding a separate virtual channel (VC) for PIM requests to alleviate congestion at the interconnect. Our analysis shows that this can improve the arrival rate of MEM requests at the memory controller by an average of 2.87x for some memory controller scheduling policies, while adding less than 5% area overheads.
- Design and evaluation of a novel memory controller scheduling policy, called F3FS, that modifies FR-FCFS by adding an extra layer of arbitration to favor current mode, but caps the number of each request type served to provide fairness. The cap can be adjusted to provide fairness between competing applications or reduce execution time for collaborating ones.

4.1 Background

4.1.1 PIM Architectures

Figure 4.1 presents an exemplar bank-level PIM architecture that integrates multiple functional units (FUs) in each HBM layer, where each FU is shared between a pair of banks. The microarchitecture of the PIM FUs is presented in Figure 4.2. Each FU incorporates a SIMD ALU along with a local register file to store operands and temporary values. The register file is DRAM word-wide, which is typically tens of bytes (32 bytes in HBM). The SIMD ALU operates on a DRAM word, which can include multiple data elements (e.g., 16 FP16 elements [108, 161]).

PIM kernels mapped to bank-level PIM architectures lay out data in row buffer-sized chunks across multiple banks, processing them in parallel one DRAM word at a time. Given the large datasets of the applications that PIM targets, this could result in a large influx of requests in a very short period. In order to curb this injection rate and provide higher command bandwidth, bank-level PIM architectures often implement a separate PIM mode [108, 161, 163, 256]. Within PIM mode, a single PIM request is

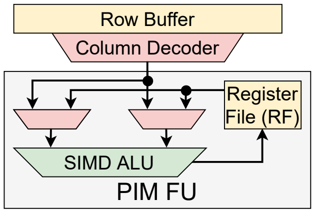


Figure 4.2: PIM functional unit (FU) microarchitecture. The SIMD ALU can implement generic math and logic operations and/or domain specific operations.

executed by all banks in a lock-step manner. The memory controller switches between PIM and MEM modes, effectively choosing which of PIM queue and MEM queue to service requests from (Figure 4.1). The PIM register file (RF) holds state across MEM/PIM switch boundaries, allowing for PIM correctness.

4.1.2 PIM Programming

There are two broad PIM programming paradigms: 1) coarse-grained offloading, where the application configures control registers in the memory controller that specify the function to compute [48, 73], and 2) fine-grained offloading, where the application issues special memory instructions that encode PIM operations (e.g., add), which are scheduled by the memory controller [162, 256]. The fine-grained instructions look and behave like non-temporal (i.e., non-cached) stores for the host core, and the model we use in this chapter.

Figure 4.3 shows an example fine-grained PIM kernel that adds two vectors. The vectors are laid out in separate rows and aligned to the row buffer size. The kernel first loads n DRAM word-sized chunks of vector $\boldsymbol{a}$ into the PIM register file. The register file contents are then added to vector $\boldsymbol{b}$, performing a DRAM word-wide SIMD operation and storing the sum into the register file. Finally, the register file contents are stored into vector $\boldsymbol{c}$.

The figure also exemplifies the *block* structure of PIM kernels, where blocks con-

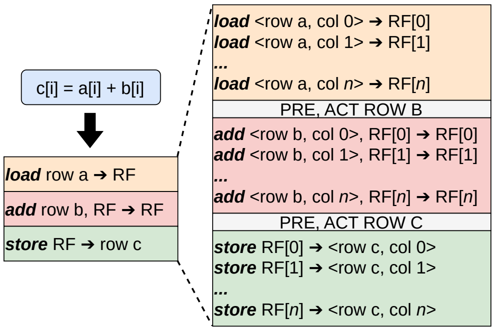

c[i] = a[i] + b[i]
load row a ➔ RF
add row b, RF ➔ RF
store RF ➔ row c
load <row a, col 0> ➔ RF[0]
load <row a, col 1> ➔ RF[1]
...
load <row a, col n> ➔ RF[n]
PRE, ACT ROW B
add <row b, col 0>, RF[0] ➔ RF[0]
add <row b, col 1>, RF[1] ➔ RF[1]
...
add <row b, col n>, RF[n] ➔ RF[n]
PRE, ACT ROW C
store RF[0] ➔ <row c, col 0>
store RF[1] ➔ <row c, col 1>
...
store RF[n] ➔ <row c, col n>

Figure 4.3: Vector addition PIM kernel. PIM kernels have a block structure, where a block consists of consecutive PIM operations to the same row. The size of the block is usually a multiple of the register file (RF) size (n).

sist of consecutive PIM operations (e.g., *load*) to the same row and are separated by a precharge and an activate. While instructions within blocks can be reordered, blocks must be executed sequentially for correctness due to their dependencies. Such sequential ordering can be achieved on the host side by using special barriers (e.g., Orderlight [197], which prevents reordering at SM's operand collector stage) and at the memory controller by using a scheduling policy like first-come first-served. We use this block structure to minimize MEM interference with PIM (Section 4.6.2).

4.1.3 Concurrent GPU Kernel Execution

Concurrent utilization of host and PIM cores is an effective way of maximizing hardware utilization and application performance. Large language models (LLMs) exploit parallelism by simultaneously computing on different fully connected layers on the host and PIM [248] and by overlapping Query/Key/Value (QKV) generation on the host with multi-head attention on PIM [111, 212]. Other domains, like graphics [290] and scientific computing [119] also achieve performance and energy gains with such concurrent execution.

While streams [13, 106, 205] can be used to launch concurrent requests from the same application, CUDA multi-process service (MPS) [207] allows for GPU resources to be shared simultaneously by different host processes. Kernels from different processes each have their own address space, but they share the GPU SMs, caches, and memory bandwidth. Taking a step further, CUDA multi-instance GPU (MIG) [206] adds the ability to *physically* partition GPU resources (including SMs, interconnect links, memory capacity, and memory bandwidth) into several *instances*, essentially creating several sub-GPUs that can be used by independent applications.

4.2 Evaluation Methodology

4.2.1 Simulator

We use a modified version of GPGPU-Sim [144] that implements a cycle-level all-bank PIM execution model, closely based on commercial designs [161]. Table 4.1 summarizes key architectural parameters. The main memory incorporates a PIM FU for a pair of banks, with each bank receiving 8 register file entries out of 16. The memory controller is updated to incorporate separate MEM and PIM queues. PIM kernels, implemented in CUDA following the ISA of the PIM architecture we model [161], send PIM operations modeled as cache streaming (CS) stores[1]. We modified GPGPU-Sim's core model to ensure that CS stores bypass all caches and are sent to the main memory directly.

4.2.2 Benchmarks

We evaluate on two application scenarios: competitive, where two separate applications launch a GPU and PIM kernel, and collaborative, where the same application launches both. The GPU and PIM kernels are launched concurrently using CUDA streams.

[1] https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#cache-operators

Table 4.1: Simulation parameters

GPU Parameters	
GPU Model: Nvidia Quadro GV100	
Number of SMs: 80	Core Frequency: 1132 MHz
L1D Cache: 32 KB	Shared Memory: 96 KB
L1I Cache: 128 KB	L2 Cache: 6 MB
Memory Parameters	
Memory Technology: HBM	
Channels/Banks: 32/16	DRAM Frequency: 850 MHz
Bus Width: 16 B	Burst Length: 2
MEM-Q/PIM-Q Size: 64 entries	NoC buffer size: 512 entries
PIM FUs: 8/channel	PIM RF Size: 16 entries
Timing Parameters (cycles):	*tCCDs*=1, *tCCDl*=2, *tRRD*=3, *tRCD*=12, *tRP*=12 *tRAS*=28, *tCL*=12, *tWL*=2, *tWR*=10, *tRTPL*=3
Address Map (bits):	RRRR.RRRRRRRR.RBBBCCCB.DDDDDCCC Key: R=Row, B=Bank, C=Column, D=Channel

In order to facilitate PIM programming, we turned off pseudo-random I-poly [225] mapping to channels in favor of a more regular scheme, listed in Table 4.1. PIM kernels use the simplified mapping to map each warp to a single memory channel and each thread within a warp to a single bank. This mapping ensures that requests to each PIM unit are issued sequentially. We use Orderlight barriers [197] to prevent reordering of requests within the SM. PIM kernels require eight SMs (total of 1024 threads, 4 warps per SM) to maximize speedup, leaving 72 SMs for the GPU kernel.

Competitive: We borrow nine PIM-amenable kernels from prior work [197][2]. Table 4.2 lists the name and input size of each benchmark. These PIM kernels are run concurrently with 20 Rodinia benchmarks [40][3] on the host GPU, giving us 180 unique

[2] STREAM-Triad was excluded because it has the same access pattern as STREAM-Add. Histogram was excluded because only a small fraction is PIM-amenable.

[3] We do not evaluate `leukocyte` because the provided input file did not generate significant memory

GPU/PIM kernel combinations. Input size for the GPU benchmarks, listed in Table 4.3, is taken from prior work [128, 135]. Both PIM and GPU kernels are run continuously in a loop until each kernel has run at least once. We report data for the first completed run of each benchmark.

Table 4.2: PIM benchmarks

No.	Benchmark	Input size
P1	Stream Add	67M elements per vector
P2	Stream Copy	
P3	Stream Daxpy	
P4	Stream Scale	
P5	BN Fwd	8M batches, with 8 elements each
P6	BN Bwd	
P7	Fully connected	Inputs = Outputs = 16, 262,144 batches
P8	KMeans	1,048,576 points, 32 features
P9	GRIM	8M bitvectors, 32 base pairs

Collaborative: We emulate the execution of a GPT-3-6.7B like large language model (LLM) [35], overlapping the execution of QKV generation with multi-head attention (MHA), similar to prior work [212]. To achieve this, we execute three GEMM kernels in a series on the GPU (QKV generation) with the PIM executing GEMV and softmax layers (MHA). The model uses a batch size of 128, sequence length of 1024, and an embedding table of size 4096. We assume that the KV cache for each layer is loaded on demand to keep the model memory footprint in check.

4.2.3 Metrics

Competitive: We compute the speedup of both the GPU and PIM kernels as the ratio of their execution time when run alone on 80 SMs and 8 SMs, respectively, to their

traffic. `hybridsort` and `particlefilter` ran for too long. `myocyte` encountered a bug with GPGPU-Sim.

Table 4.3: GPU benchmarks

No.	**Benchmark**	**Input size**
G1	b+tree	1 million keys, 10000 bundled queries, a range search of 6000 bundled queries with the range of each search 3000
G2	backprop	655360 input nodes
G3	bfs	1 million vertices
G4	cfd	97K elements
G5	dwt2d	1024x1024 images, forward 5/3 transform
G6	gaussian	2048x2048 matrix
G7	heartwall	656x744 video, 2 frames
G8	hotspot	2048x2048 data points, pyramid height=4, 2 iterations
G9	hotspot3D	512x512 data points, 8 layers, 10 iterations
G10	huffman	262144 elements
G11	kmeans	494020 points, 34 features
G12	lavaMD	1000 boxes
G13	lud	2048x2048 data points
G14	mummergpu	Reference: 20K sequences, 71 characters; Query: 50K sequences, 25 characters
G15	nn	10000390 hurricanes across 10 files, 10 nearest neighbors
G16	nw	2048x2048 data points
G17	pathfinder	100000x100 grid, pyramid height=4
G18	srad_v1	512x512 data points, 100 iterations, lambda=0.5
G19	srad_v2	2048x2048 data points, 2 iterations, lambda=0.5
G20	streamcluster	65536 points, 256 dimensions, 10-20 centers, 1000 intermediate centers

execution time when run under contention. We then use this speedup to evaluate each scheduling policy across two key metrics: fairness and throughput. Fairness is defined using Fairness Index [72] which quantifies the disparity between the individual GPU and PIM kernel speedups. It is expressed as:

$$\textit{Fairness Index} = min(\frac{Speedup_{PIM}}{Speedup_{MEM}}, \frac{Speedup_{MEM}}{Speedup_{PIM}}) \tag{4.1}$$

Throughput is defined using System Throughput [72] which quantifies the kernel execution rate of the system, measured as the sum of speedups of the GPU and PIM kernels. This is a direct measure of concurrency and the rate at which the system can service kernels.

Collaborative: The key metric in this scenario is the speedup of the concurrent kernel execution relative to the serial execution of the kernels. We compare this speedup to an ideal scenario where the total execution time is the execution time of the longer running kernel when run alone, representing perfect overlap.

4.2.4 Memory Controller Scheduling Policies

We summarize below the baseline memory scheduling policies we evaluate. Note that most of these policies were not designed for PIM architectures; we therefore explain how they switch between PIM and MEM modes.

1. *First-Come First-Served (FCFS)*: Executes requests in the order they arrive. Switches modes according to the request type.

2. *MEM-First*: Always issues MEM requests, if there are any. Prior art has used this policy before [48].

3. *PIM-First*: Always issues PIM requests, if there are any.

4. *First-Ready FCFS (FR-FCFS)* [235]: Prioritizes row buffer hits over the oldest request, switching modes if the oldest request is from a different mode at the time of a row buffer conflict on all banks. Each bank maintains a conflict bit, which is set when there is a row buffer conflict and the oldest request is from a different mode. The bank then stalls until a mode switch occurs, which is performed after every bank has set its conflict bit.

5. *FR-FCFS-Cap:* A fairer version of FR-FCFS that CAPs the number of row buffer hits that bypass the oldest request [193].

6. *Blacklisting Memory Scheduler (BLISS)* [261]: Blacklists applications that issue more than *n* requests consecutively under FR-FCFS. Then implements the following priority order: 1) non-blacklisted application first, 2) row buffer hit first, 3) oldest first. The blacklist is cleared every few thousand cycles. This mechanism effectively deprioritizes high memory intensity applications.

7. *First-Ready Round-Robin FCFS (FR-RR-FCFS)* [137]: Modifies FR-FCFS to improve fairness by cycling through modes on row buffer conflicts, effectively implementing the following priority order: 1) row buffer hit first, 2) next mode in round-robin order first, 3) oldest first within the current mode.

8. *Gather & Issue (G&I)* [162]: Switches to PIM when PIM queue occupancy reaches a *high* watermark, then drains the queue until the occupancy falls below a *low* watermark.

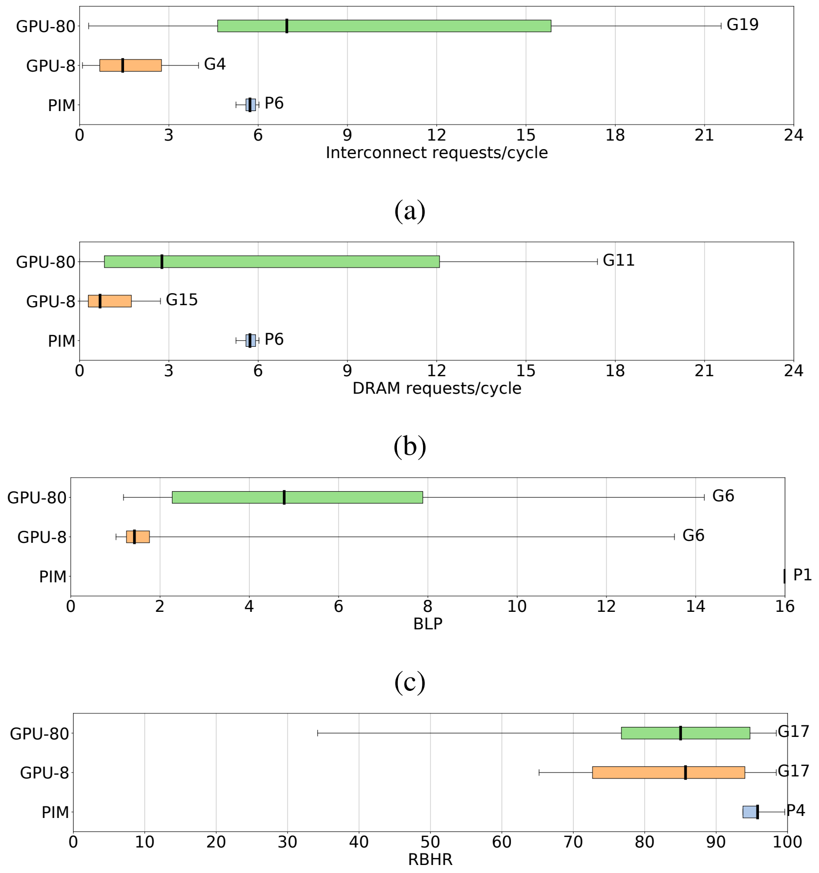
Each of the above described policies use FR-FCFS within MEM mode, except FCFS, while PIM requests always execute in FCFS order to ensure correctness.

4.3 Characterizing GPU/PIM Interference

In order to understand the performance impact of co-executing PIM and GPU kernels, we first quantify each application type's memory behavior. Figure 4.4 compares the memory access characteristics of the Rodinia benchmark suite to the PIM kernels under FR-FCFS policy, in terms of (a) interconnect request arrival rate, (b) DRAM request arrival rate, (c) DRAM bank-level parallelism (BLP), and (d) DRAM row buffer hit-rate (RBHR). The boxes represent the inter-quartile range for each metric, with the middle line and whiskers representing the median and extremes, respectively. Request arrival rates are measured in terms of total GPU cycles, while BLP is measured in terms of active DRAM cycles, i.e., the average BLP while the DRAM is servicing at least one request. Since PIM kernels only need eight SMs to fully saturate the memory subsystem interface, we compare them to Rodinia kernels running on both 80 and 8 SMs (represented as GPU-80 and GPU-8, respectively).

PIM kernels have a 3.95x higher request arrival rate into the interconnect compared to GPU-8, and is only 17.8% lower than GPU-80, on average. While regular memory requests get filtered by the L2 cache, PIM requests are not, worsening the imbalance at the memory controller. PIM request arrival rate at the memory controller is heavier than both GPU-8 *and* GPU-80, on average, outpacing them by 8.33x and 2.07x, respectively.

Not only can MEM and PIM requests not be issued concurrently, but they also exhibit very different memory access behavior. Figures 4.4c and 4.4d compare the BLP and the RBHR of the GPU and PIM kernels. PIM kernels not only execute on all banks at the same time (Figure 4.4c shows a single bar at 16 for PIM kernels), but also exhibit high row buffer locality. Combined with their high request arrival rate (Figure 4.4b), PIM kernels can severely affect a co-executing application's performance. Figure 4.5 compares the impact of memory intensive GPU kernels and PIM kernels on co-executing kernels.



(a)

(b)

(c)

(d)

Figure 4.4: Memory access characteristics of the Rodinia benchmark suite, running on 80 and 8 SMs, and the PIM kernels in terms of (a) interconnect request arrival rate, (b) DRAM request arrival rate, (c) DRAM bank-level parallelism (BLP), and (d) DRAM row buffer hit rate (RBHR). The high whiskers are labeled with the most intensive kernel for that metric.

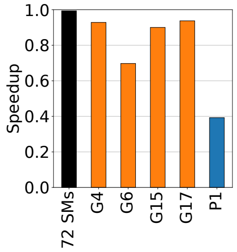
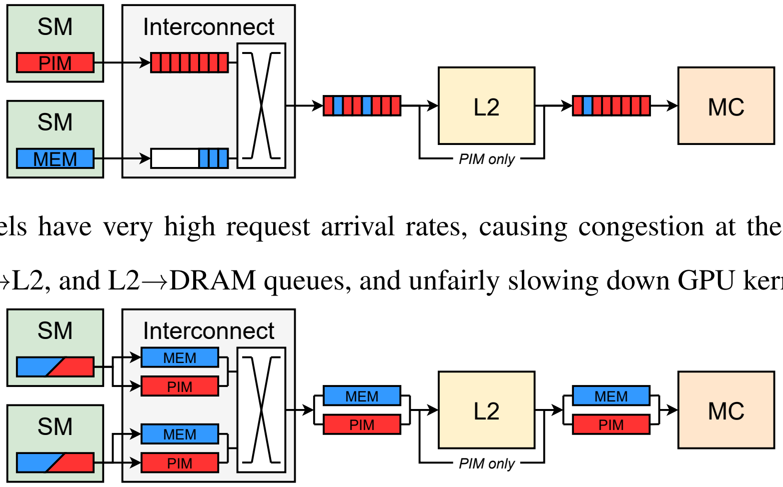


Figure 4.5: Average speedup of Rodinia benchmark suite when running on 72 SMs and when co-executing with four memory intensive kernels.

The figure shows the average speedup of the Rodinia benchmark suite running on 72 SMs, with the remaining 8 SMs occupied by one of G4, G6, G15, G17, and P1. The four chosen GPU kernels are the most memory intensive in terms of interconnect requests (G4), DRAM requests (G15), BLP (G6), and RBHR (G17) when running on 8 SMs (Figure 4.4c). PIM kernels show very little variation across each metric and so we picked P1. The speedup is normalized to Rodinia benchmark suite running alone on 80 SMs. To separate the effects of memory contention and reduced SM availability, the figure also presents the speedup of running the kernels on 72 SMs without any contention. The figure shows how the benchmark suite slows down by an average of 60% when co-executing with P1, compared to a worst-case average slowdown of 30% when running with other Rodinia kernels.

4.4 Memory Access: Interconnect Bottlenecks

PIM kernels are optimized to maximize the utilization of PIM FUs by saturating the memory subsystem. This leads to a very high request arrival rate for the duration of the kernel's execution that can deny service to co-executing applications (Figure 4.4a). Figure 4.6a shows this scenario, where the PIM requests fill the interconnect→L2 and L2→DRAM queues, denying service to GPU kernels. To quantify this degradation, Figure 4.7a characterizes the request arrival rate of each GPU kernel under each memory scheduling policy, averaged across all PIM kernels. We present results for each scheduling policy since the service rate of each policy determines how fast the PIM requests are drained from the interconnect. Note that some applications experience an increase in the arrival rate. This is because PIM interference increases MEM queuing delay, improving MEM RBHR and reducing GPU kernel's overall execution time.



(a) PIM kernels have very high request arrival rates, causing congestion at the interconnect, interconnect→L2, and L2→DRAM queues, and unfairly slowing down GPU kernels.

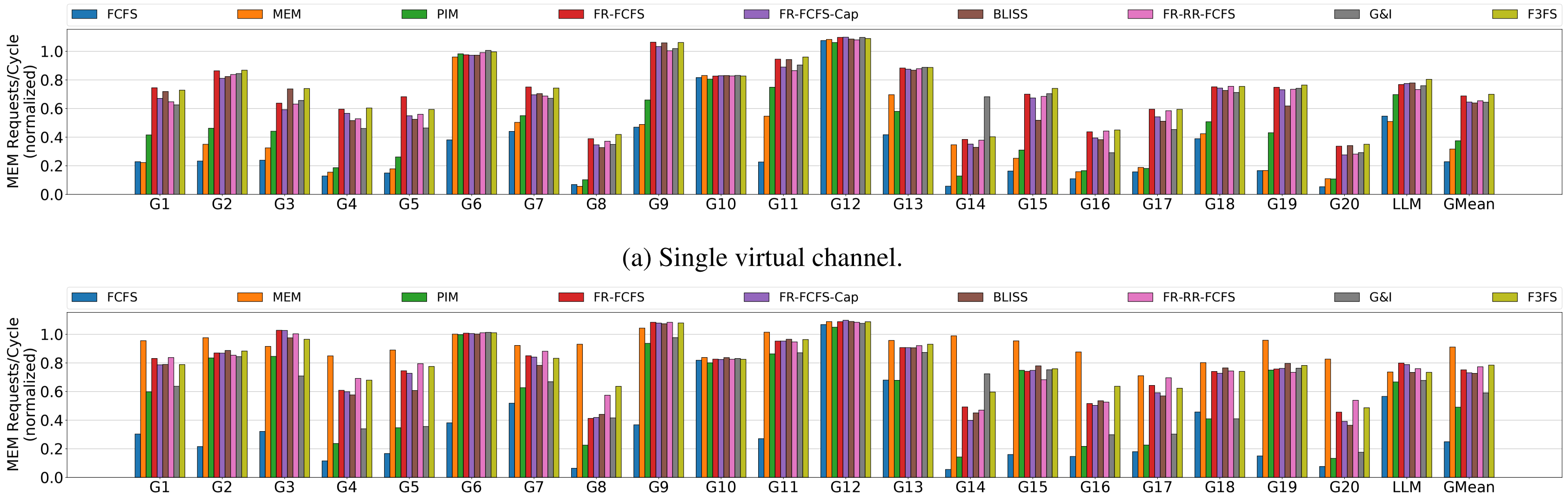
(b) Separation of MEM (blue) and PIM (red) requests into separate virtual channels and queues to minimize interference between them.

Figure 4.6: Comparison of the baseline memory subsystem (a) with our proposed changes (b).

While throughput optimizing policies like FR-FCFS are able to sustain a higher arrival rate for MEM requests than others, the degradation remains severe, with even FR-FCFS suffering a 41% drop on average. A policy like MEM-First should, intuitively, perform well here, but its performance is limited by the fact that most MEM requests are stalled behind PIM requests in the interconnect. This demonstrates that even though the memory controller scheduling policy impacts interconnect congestion, PIM kernels' memory intensity necessitates changes to the interconnect architecture.

4.4.1 Separating MEM and PIM Virtual Channels

In order to alleviate the problem of congestion at the interconnect, we propose separating MEM and PIM requests into separate queues all the way from the SMs to the memory controller. Figure 4.6b illustrates this proposal. Memory requests entering the interconnect from the SMs are split into two virtual channels (VCs), one each for MEM



(a) Single virtual channel.

(b) Separate virtual channels.

Figure 4.7: MEM request arrival rate into the memory controller, without (a) and with (b) separate MEM and PIM virtual channels, normalized to standalone execution (higher is better).

and PIM requests. These virtual channels empty into interconnect→L2 cache queues, where MEM requests are picked up by the cache while PIM requests are forwarded to the memory controller. Finally, the two request types also share the links between L2 cache and memory controllers, necessitating splitting of L2→DRAM queues as well.

Using separate VCs and queues ensures that the two request types do not interfere until they reach the memory controller, preventing PIM requests from stalling MEM requests. Furthermore, the system provides fairness at each link by switching between the two queues in a round-robin fashion. The crossbar interconnect uses a modified version of the iSlip algorithm [187] where the arbiter records the previous VC served for each incoming link and switches to the other VC presuming there is traffic on it. The efficacy of this solution is demonstrated in Figure 4.7b. We split existing interconnect queues in half to add a PIM VC, keeping the *total* queue size in Figures 4.7a and 4.7b equal. While most policies experience an increase in the arrival rate, MEM-First experiences the biggest jump, with its average degradation reducing from 68% to 9% (2.87x improvement).

Adding virtual channels entails area and power costs, however. The VC allocator, used for allocating output virtual channels to input virtual channels, grows quadratically in the number of ports and virtual channels [300]. Meanwhile, the number of control wires to encode the VC information with each packet grows logarithmically. Despite the significant asymptotic growth, additional VCs add modest area and power overheads, especially if the queues are long and the routers are pipelined. Based on the data from Yoon et al. (Figure 4(c) in [300]), a router based on 45nm technology with 32 queue entries, 128-bit channels, and a clock speed of 1ns experiences ∼5% increase in area when going from a single VC to two. Since our evaluated system is based on a smaller process node (12nm), uses longer queues (512 entries), and runs at a comparable clock speed (0.88ns), we expect the overheads to be even lower.

4.5 Memory Utilization: Scheduling Bottlenecks

Past the interconnect queues, MEM and PIM requests again contend at the memory controller. The memory controller needs a *mode switching policy* to switch between MEM and PIM modes to serve requests of each type. The design of an efficient mode switching policy is non-trivial, for two reasons. The first reason is the increase in queuing delays. Since MEM and PIM requests cannot execute concurrently, each of them suffers increased queueing delays while waiting for requests of the other type to complete. However, each application has a different tolerance for queueing delays, creating a fairness problem.

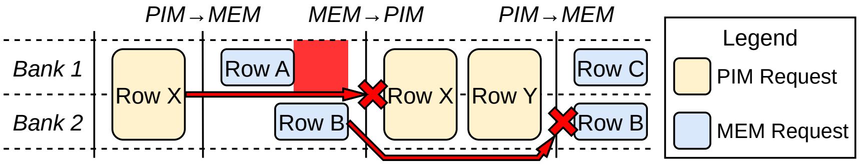


Figure 4.8: Switching between MEM and PIM modes leads to loss in locality since the two request types often map to different rows, as seen for requests mapping to Rows X (PIM) and B (MEM). MEM→PIM switches also suffer from bank idle time, like Bank 1 in the figure, since MEM requests on different banks execute asynchronously.

The second difficulty is the non-trivial cost of switching, as depicted in Figure 4.8. When the memory controller performs a MEM→PIM switch, all in-flight MEM requests must be drained before any PIM request can be issued. Since each memory bank services requests concurrently and independently, this leads to idle time for banks that finish first (Bank 1 in Figure 4.8). In addition, both MEM→PIM and PIM→MEM switches may cause reduced row buffer locality since the two types of requests often map to different rows (Rows X and B in Figure 4.8).

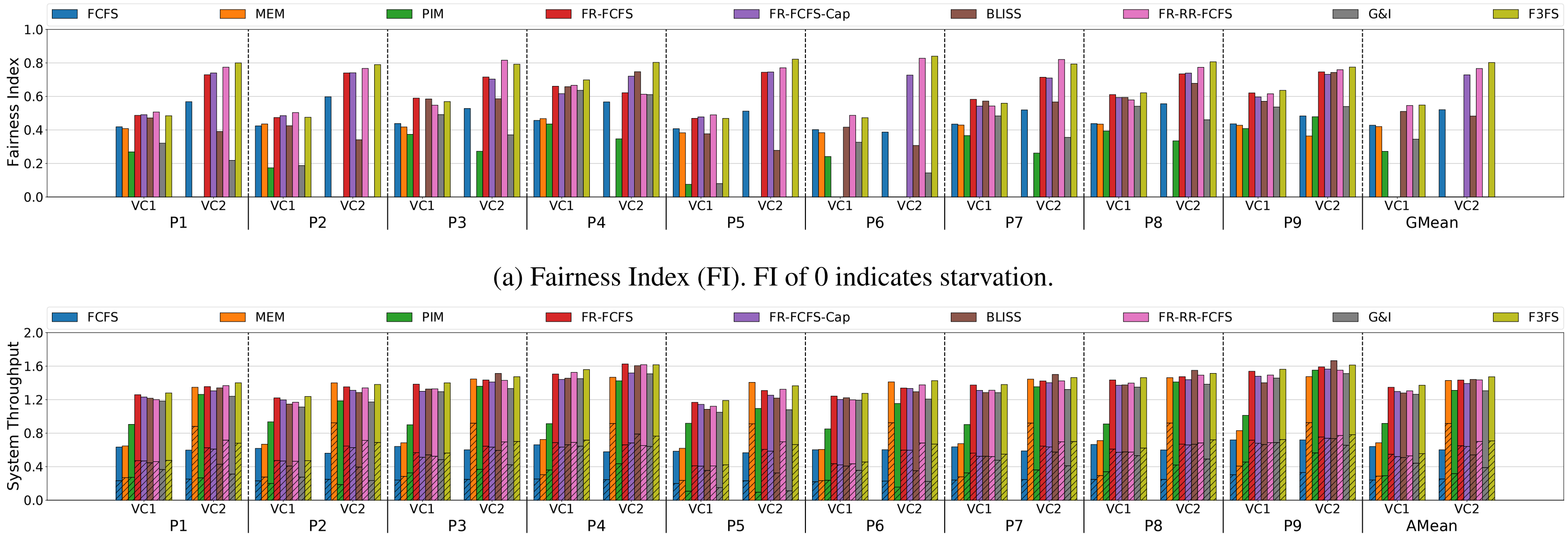
4.5.1 Competitive Co-execution

We first characterize the performance of the various memory controller scheduling policies that we listed in Section 4.2.4 under the competitive scenario, both with and without separate MEM/PIM VCs. We label the two configurations as VC1 (Figure 4.6a) and VC2 (Figure 4.6b). Figure 4.9 shows the fairness index and throughput for each PIM kernel, averaged across all GPU kernels. Figure 4.10 presents the average (a) number of mode switches (normalized to FCFS), (b) number of additional MEM conflicts per switch, and (c) latency of draining MEM queue per switch, across all combinations. Figure 4.10a uses geometric mean, while 4.10b and 4.10c use arithmetic mean. The figures also include results for our proposed policy, which we will introduce and discuss later in Section 4.6.

FCFS schedules memory requests in arrival order, resulting in frequent switches (Figure 4.10a). As a result, both individual application performance and hardware utilization can be compromised. While such switching helps fairness to a certain degree, especially with the VC2 configuration, the lack of locality and parallelism awareness hurts throughput.

MEM-First and **PIM-First** favor a single request type, with the potential for the other request type to experience extreme unfairness or starvation: a fairness index of 0 is common (Figure 4.9a). Most throughput gains often stem from a single application that submits the request type favored by the policy (Figure 4.9b). Both policies also suffer from frequent switching (Figure 4.10a) and high switch overheads (Figures 4.10b and 4.10c), particularly in the VC1 configuration.

FR-FCFS optimizes for locality by prioritizing row buffer hits over older requests. Such a design introduces two sources of unfairness: 1) high row buffer locality, and 2) high access frequency. Both characteristics are true of PIM kernels (Section 4.3), meaning that FR-FCFS inherently favors PIM kernels. This is evident from Figure 4.9b, where MEM speedup contributes as little as 35% to the overall speedup (P5/VC1,

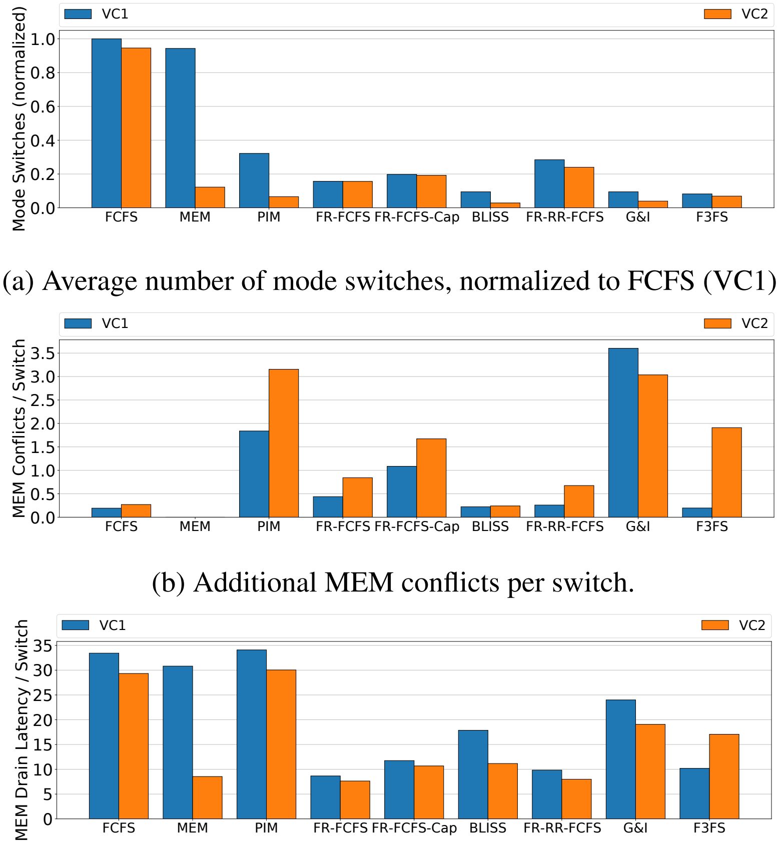


(a) Fairness Index (FI). FI of 0 indicates starvation.

(b) System Throughput. The shaded and non-shaded regions represent MEM and PIM speedups, respectively.

Figure 4.9: Fairness (a) and throughput (b).

P6/VC1), with the average contribution at 41% and 45% with VC1 and VC2 configurations, respectively. **FR-FCFS-Cap** solves the first unfairness issue by restricting the number of row buffer hits that bypass older requests, improving fairness and providing starvation freedom with the VC2 configuration. The policy still, however, suffers from the second source and can cause starvation with VC1 configuration (P3, P6). The CAP, set empirically to 32, also introduces more switches (Figure 4.10a), which reduces throughput by 3.7% and 2.8%, on average, under VC1 and VC2, respectively, compared to FR-FCFS.



(a) Average number of mode switches, normalized to FCFS (VC1).

(b) Additional MEM conflicts per switch.

(c) MEM drain latency (in DRAM cycles) per switch.

Figure 4.10: Average number of mode switches (a) and MEM→PIM switch overheads in terms of additional MEM conflicts (b) and the latency of draining the MEM queue (c).

BLISS builds upon FR-FCFS, but performs worse than it in both our key metrics. Depending on its blacklist schedule, BLISS devolves into either one of MEM-First, PIM-First, or FR-FCFS. Our analysis shows that it spends around 20%, 20%, and 60% time in each of those states with a blacklist threshold of 4. Performing a sweep of the blacklist threshold, we note that BLISS performs best with a lower threshold, indicating its tendency to converge toward FR-FCFS.

G&I, while designed to be a MEM-friendly policy [162], heavily favors PIM requests and even causes starvation with the VC2 configuration (Figure 4.9a, P5). Owing to PIM kernels' high request arrival rate, PIM requests cross the *high* threshold (set to 56) very quickly and fall below the *low* threshold (set to 32) only when either: 1) MEM requests create back pressure, or 2) the PIM kernel is nearing completion. VC2 mitigates interference at the interconnect, leading to starvation similar to MEM-First and PIM-First.

FR-RR-FCFS is the fairest policy in our characterization, achieving average fairness indices of 0.55 and 0.77 with VC1 and VC2 configurations, respectively. By switching mode on row buffer conflicts, FR-RR-FCFS ensures that all co-executing applications receive service and resolves the FCFS-inherent unfairness in FR-FCFS. However, the policy is sill prone to favoring applications with high locality since only row buffer conflicts switch the application being serviced. We see this in Figure 4.9b with P4/VC2, where the PIM kernel's (STREAM-Scale) high locality (99.6%) hurts GPU kernels' speedup and provides 60% of the throughput, on average.

4.5.2 Collaborative Co-execution

We next look at the collaborative scenario where all policies execute a decoder-only LLM (Section 4.2). This scenario is different from the competitive scenario in that the primary metric is total execution time, not fairness. Figure 4.11 presents the speedup under each policy compared to sequential execution of the two kernels. Under VC1 configuration, all policies struggle to achieve any speedup. The key problem here is that QKV generation, running on GPU SMs, is the longer running of the two kernels, but the PIM kernels produce significantly more traffic and restrict the time MEM requests receive service. This allows a policy like G&I to work well. By favoring PIM requests, G&I is able to drain the interconnect and reduce congestion, allowing MEM requests to make progress. At the memory controller, MEM requests are issued once the MEM queue fills up and prevents PIM requests from coming in. While PIM-First should exhibit similar characteristics, it lags behind because of nearly double the number of switches and 6x higher switching latency.

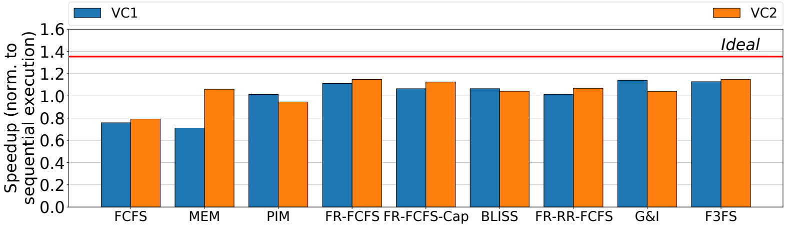


Figure 4.11: LLM speedup for each policy with both a combined and separate VCs. The speedup is normalized to sequential execution of QKV generation and multi-head attention, while the *Ideal* represents the minimum of the two stages.

The results vary significantly in VC2 configuration. Since MEM and PIM requests do not interfere at the interconnect, PIM-favoring policies like G&I perform very close to sequential execution. MEM-First achieves a speedup >1 because it favors the slower running kernel, but it still limits parallelism. FR-FCFS shines here because it is able to maximize memory throughput, minimizing the overall execution time.

4.6 First Mode-FR-FCFS (F3FS) - New and Improved PIM-Aware Memory Access Scheduling

We propose a new memory controller scheduling policy, called **First Mode-FR-FCFS** (shortened to **F3FS**), which attempts to improve both fairness and throughput. F3FS adds a new arbitration stage in front of FR-FCFS that favors requests in the current mode. That is, it implements the following priority order: 1) current mode first, 2) row buffer hit first, 3) oldest first. Within MEM and PIM modes, the queues utilize FR-FCFS and FCFS, respectively. By favoring requests in the current mode, F3FS improves throughput by maximizing locality and minimizing the switching frequency. To prevent one mode from starving another, F3FS also implements a *CAP* on the number of requests serviced in the current mode that bypass an older request in the other mode. Here, age is implemented as an incrementing ID assigned to each request as it enters the memory controller.

While fairness is an important metric for competitive co-execution that favors equal CAPs on MEM and PIM requests, collaborative co-execution may favor an unequal split of resources that results in an overall lower execution time. To support this, F3FS uses two CAPs, one each for MEM and PIM modes. In a collaborative scenario, the application can favor one type of kernel by setting a higher CAP value for it than the other. These asymmetric CAPs can also be configured by system software to enforce process priorities in competitive scenarios. We leave an exploration of the latter to future work.

4.6.1 Hardware Implementation

Figure 4.12 presents the architecture of the mode switch logic. In particular, the figure highlights the additions, deletions, and modifications for F3FS compared to the FR-FCFS switching policy. While F3FS introduces additional comparators and structures for counting the number of bypasses, it also gets rid of the per-bank conflict tracking that FR-FCFS performs. Such tracking goes beyond just maintaining a single bit for each bank and implementing the AND circuitry: for instance, the logic needs to track whether every bank has had at least one request issued before marking the next request as a conflict.

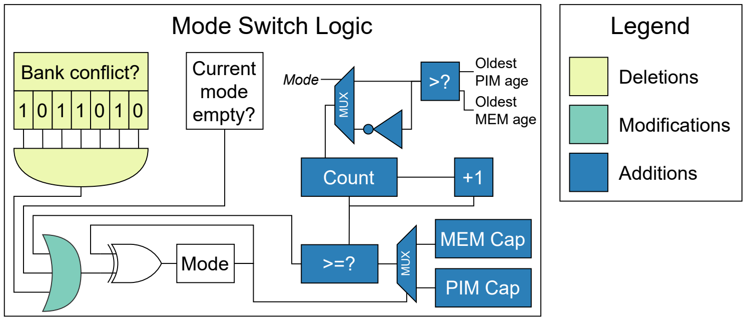


Figure 4.12: Hardware overheads of F3FS in terms of the mode switch logic complexity, compared to FR-FCFS.

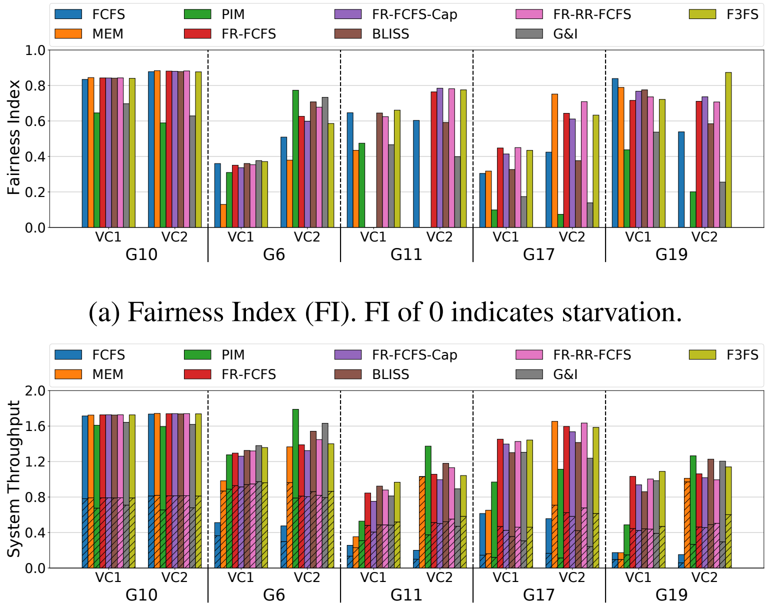
In order to quantify the area overheads of F3FS over FR-FCFS, we synthesized their mode switching logic on an AMD XCZU5EV FPGA [15] using Vitis HLS [14]. The synthesis reveals that F3FS requires 275/143 LUTs/flip-flops, compared to 377/88 for FR-FCFS.

4.6.2 Evaluation

We compare F3FS with the baseline policies under both the VC1 and VC2 configurations. First, we look at competitive co-execution, where we use the same CAP (empirically set to 256) for both MEM and PIM to promote fairness. This CAP, determined from a sensitivity study, is strategically set to a multiple of the PIM RF size (eight per bank) to exploit the block structure of PIM kernels (Section 4.1.2). Figure 4.9 shows the fairness and throughput improvements.

Competitive co-execution: F3FS performs the same or better than the best performing state-of-the-art policies in both VC1 and VC2 configurations. Under VC1 configuration, F3FS provides fairness comparable to FR-RR-FCFS, while achieving 1.8% and 5.1% higher average throughput compared to FR-FCFS and FR-RR-FCFS, respectively. This is a direct result of F3FS switching less frequently (Figure 4.10a) while paying comparable costs per switch (Figures 4.10b and 4.10c). Meanwhile, under VC2 configuration, F3FS outperforms FR-RR-FCFS in terms of both fairness and throughput by 4.7% and 2.6%, respectively, on average. This is a key result that highlights the throughput benefits of favoring current mode and fairness benefits of capping the wait time of each mode. Beyond averages, F3FS improves the worst-case fairness/throughput by 76.76%/28.98% under VC1 and by 146.22%/29.84% under VC2, respectively, compared to FR-RR-FCFS. Combining F3FS with our proposed interconnect changes yields average and best case fairness/throughput improvements of 48%/13% and 72%/22%, respectively, compared to a baseline single VC interconnect and FR-RR-FCFS policy. These improvements highlight how F3FS enhances the feasibility of concurrent host/PIM execution.

In order to evaluate how extremes in the memory intensity of the GPU kernels affects the performance of F3FS in a competitive scenario, Figure 4.13 shows the average fairness and throughput when a PIM kernel is executed with a compute intensive kernel (G10) or one of four of the most memory intensive Rodinia kernels (G6, G11, G17, G19), averaged across all PIM kernels (an orthogonal slice of Figure 4.9). The memory intensive kernels are picked based on our characterization in Figure 4.4. With the compute intensive kernel G10, there is very little variation in both fairness and throughput across scheduling policies and interconnect configurations, highlighting such applications' tolerance for memory access delays. Memory intensive kernels have more varied results. F3FS works well with G19, where interconnect traffic is high, but is filtered by the L2 cache, and is indicative of the common case of moderate memory traffic. F3FS is able to equalize queuing delays for MEM and PIM requests by using a symmetric CAP, while maintaining long enough phases to achieve BLP and RBHR comparable to standalone execution and minimize switching overheads. With G6 and G11, F3FS



(a) Fairness Index (FI). FI of 0 indicates starvation.

(b) System Throughput. The shaded and non-shaded regions represent MEM and PIM speedups, respectively.

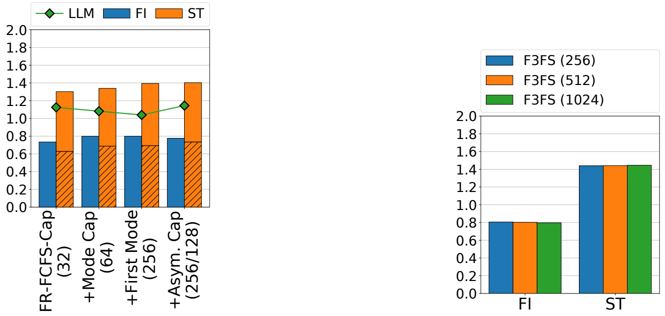
Figure 4.13: Fairness (a) and throughput (b) of a compute intensive (G10) and four memory intensive (G6, G11, G17, G19) Rodinia kernels, averaged across all PIM kernels.

unfairly favors GPU kernels due to long MEM phases. G6 achieves higher BLP with F3FS than other policies, elongating the MEM drain latency per MEM→PIM switch. G6 also has long MEM phases because of its poor locality (average RBHR of 32%). F3FS, by equalizing the number of MEM and PIM requests served, inadvertantly leads to longer MEM phases because MEM requests take longer than PIM requests, on average. G11's high MEM request arrival rate ensures that MEM requests frequently execute up to the CAP. Even when the CAP is reached, the high MEM arrival rate often results in staying in MEM mode due the oldest request at the memory controller continuing to be MEM. G17's high RBHR results in smaller MEM phases, resulting in unfairly high PIM speedup due to the application's sensitivity to prolonged MEM queuing delays resulting from a PIM CAP of 256.

Collaborative co-execution: Figure 4.11 presents the speedup of the LLM under each policy. This is where F3FS's asymmetric CAP comes into play. We configure F3FS to use MEM/PIM CAPs of 256/128 and 64/64 in VC1 and VC2 configurations, respectively, based on a sensitivity study. Setting the asymmetric CAPs is a balancing act since throughput favors high CAPs while fairness favors lower ones. Starting with high values, the asymmetric CAP under VC1 configuration is lowered based on two principles: 1) a high enough PIM CAP to ensure consistent influx of MEM requests into the memory controller, and 2) a high enough MEM CAP to service as many MEM requests as possible without starving PIM requests. While this asymmetry helps in the VC1 configuration, VC2 configuration favors a symmetric CAP. To understand this, we discuss the MEM and PIM CAPs separately. For MEM CAP, increasing the value beyond 64 did not help since only 8% of MEM→PIM switches were triggered due to the CAP being exceeded. Meanwhile, for the PIM CAP, lowering the value below 64 had two implications: 1) increased switch overheads, and 2) fewer MEM requests in the MEM queue at the end of a PIM phase, reducing the memory controller's visibility into the GPU kernel's memory access stream and hurting locality. The chosen parameters allow F3FS to match the best performing policies in both the configurations (G&I in VC1 and FR-FCFS in VC2). Compared to FR-RR-FCFS, F3FS improves speedup by 11.23% and 7.37% in VC1 and VC2, respectively. These results highlight the flexibility of F3FS, showing how it can be dynamically configured to an application's needs.

4.6.3 Discussion

Ablation study: In order to better understand F3FS's performance improvements, we study the impact of its three components that differ from FR-FCFS-Cap: 1) CAP on the number of requests serviced in current mode (vs. row buffer hits), 2) prioritizing current mode first, and 3) the ability to use asymmetric CAPs on MEM and PIM modes. Figure 4.14a shows the incremental performance impact of the three components under VC2 configuration for P2 (averaged across all GPU kernels, except `kmeans` since it starves with FR-FCFS-Cap) and the LLM. The CAP for each stage is set separately to maximize competitive performance, and is listed in the figure. Moving the CAP from limiting row buffer hits to limiting requests in the current mode improves the average fairness index from 0.73 to 0.8 for P2, while reducing the LLM speedup by 4%. Next, favoring current mode brings about throughput improvements by reducing the number of switches, while still maintaining nearly the same fairness index. The LLM, on the other hand, drops to a speedup of 1.04. Finally, the last bar demonstrates the impact of asymmetric CAPs (MEM/PIM CAPs of 256/128 respectively to prioritize the slower MEM kernel). Asymmetry negatively impacts fairness in a competitive scenario, but benefits the LLM by reducing the queuing delay for MEM requests, improving speedup by 10% and pushing it higher than that of FR-FCFS-Cap.



(a)

(b)

Figure 4.14: (a) Impact of F3FS components on fairness index (FI) and system throughput (ST) of P2 and speedup of the LLM. The shaded and non-shaded ST regions represent MEM and PIM speedups, respectively. (b) Sensitivity of F3FS to interconnect queue size under VC2 configuration in terms of FI and ST across all GPU/PIM combinations.

Sensitivity to interconnect queue size: Figure 4.14b shows the performance sensitivity of F3FS to the interconnect queue size. The queue size is varied from half (256) to double (1024) the baseline size of 512. The figure shows how F3FS is largely agnostic to the queue size itself and neither benefits from longer nor is impeded by shorter queues.

4.7 Related Work

PIM integration: UM-PIM [309] is a hardware/software memory management scheme that optimizes accesses to PIM-enabled memory and traditional memory for locality and bandwidth, respectively. PIM-MMU [160] optimizes data transfers between PIM-enabled and traditional memory by using a dedicated Data Copy Engine that incorporates a PIM-aware memory scheduling policy. Like UM-PIM, PIM-MMU also uses separate optimized memory mapping schemes for each memory type. PyPIM [165] is an end-to-end programming framework for memristive PIM [54, 260] that incorporates a PIM ISA, a host driver, and a Python development library, along with a GPU-accelerated PIM simulator for testing and validation. PIM-Enabled Instructions (PEI) [7] is a locality-aware PIM offloading framework that utilizes ISA extensions to program an in-core PEI Computation Unit (PCU). The PCU executes the instructions either locally or on PIM based on the locality of its input operands. GraphPIM [198] is another PIM offloading framework designed for graph workloads. GraphPIM works by offloading atomic instructions that access an uncacheable PIM memory space to PIM units.

Memory controller scheduling: Memory request scheduling for CPUs in multi-application scenarios is a well-studied problem [147, 148, 193, 194, 261]. STFM [193] equalizes memory-related slowdowns across competing threads by tracking the L2 stall times for each competing thread. PAR-BS [194] optimizes for fairness by scheduling at batch granularity and limiting the number of requests from each thread in a batch. TCM [148] also targets fairness by separating latency sensitive and bandwidth sensitive threads based on their LLC misses per kilo instructions, prioritizing the former over latter. STFM [193], PAR-BS [194], and TCM [148] optimize for fairness among threads, but require expensive state maintenance and communication that is untenable for GPUs. The latter two are also known to be too complex and slow for modern high speed memories [261]. ATLAS [147] is a throughput-optimizing policy that forms a total order of threads based on traffic from *all* memory controllers, prioritizing threads that have attained the least service. Like the previous three, ATLAS suffers from high complexity. ITS and WEIS [138] are instruction throughput and weighted speedup optimizing policies that utilize LLC misses per kilo instructions and DRAM bandwidth, respectively, to prioritize applications. Both would devolve into MEM/PIM-First depending on their priority order. DASH [271] is a memory scheduler for accelerator-rich systems that provides quality-of-service to hardware accelerators executing real-time applications while ensuring CPU applications make progress whenever possible.

SMS [27] is a memory scheduling policy for shared DRAM CPU/GPU systems, where GPUs are often significantly more memory intensive than CPUs. While SMS works in the presented context, its batch granularity scheduling makes it unsuitable for concurrent host/PIM accesses. In particular, CPU/GPU batches that map to different banks can be serviced in parallel, but host/PIM batches can not. SMS does not take this exclusivity in account. G&I [162] is also a PIM-aware policy for bank-level PIM architectures. Our evaluation shows how the policy is PIM-biased and is outperformed by F3FS.

Host/PIM concurrency: LLMs can leverage host/PIM concurrency by overlapping QKV generation and MHA on host and PIM, respectively (NeuPIMs [111], AttAc! [212]) or by distributing fully connected layers between host and PIM at a head granularity (IANUS [248]). While AttAc! and IANUS take care to not submit MEM and PIM requests simultaneously, NeuPIMs proposes a dual row buffer architecture, one each for serving MEM and PIM requests. F3FS makes none of these assumptions and can be tuned based on application characteristics. Pimacolaba [119] proposes software and hardware optimization to parallelize Fast Fourier Transforms across GPU SMs and PIM FUs. Chopim [48] optimizes data layout and mapping to main memory, along with OS page coloring, to reduce PIM/host access interference and maximize main memory utilization. Pattnaik et al. [215] combine a GPU vs. PIM affinity model with a dynamic execution time prediction model to dispatch GPU kernels to GPU cores or PIM FUs, with the goal of minimizing overall execution time. Most application scheduling and memory management schemes can be combined with our proposals to improve system utilization.

4.8 Summary

Integrating PIM-enabled main memories into existing systems remains an open challenge. In this chapter, we show how the sharing of the interconnect and main memory controller by *MEM*ory and *PIM* requests can severely degrade application-level fairness and system-level throughput. The memory intense nature of PIM kernels can overwhelm the memory controller under contention and create back pressure in the interconnect, hurting any co-executing application's memory performance. Our characterization of a GPU/PIM system shows that such contention causes MEM request arrival rate at the memory controller to drop by up to 95%. We propose a two-step solution to remedy this. First, we modify the interconnect and add a separate virtual channel (VC) for PIM requests to mitigate MEM/PIM interference. Second, we introduce a new memory controller scheduling policy, called F3FS, that improves: 1) fairness, by providing equal service to the two request types, and 2) throughput, by minimizing MEM/PIM mode switching frequency. When evaluated on 180 competing GPU/PIM kernel combinations, our solution achieved up to 72% and 22% better fairness and throughput, respectively, compared to FR-RR-FCFS policy with a single VC. F3FS is also tunable at runtime and can be configured to favor one request type over another, allowing a GPT-3 like LLM to execute 13.14% faster when compared to the same baseline. Beyond performance averages, F3FS also improves worst-case throughput and fairness metrics, enhancing the feasibility of GPU/PIM co-execution.

5 On The Impact of Emerging Heterogeneous Memory on Accelerator Performance

The memory footprint of modern applications like large language models (LLMs) continues to grow at a staggering rate, with model parameter size increasing by 410x every two years [87]. The growth in size and complexity has led to the wide proliferation of AI-enabled applications, from chat bots [238] and coding assistants [77] to impacting "literature and medicine" [36].

While massively parallel processors like GPUs continue to underpin the development of LLMs, their compute capacity remains underutilized [87, 142, 213] with memory capacity emerging as the key bottleneck. The challenge of accommodating these models in memory brings a long known problem to the forefront, that of DRAM capacity scaling [192]. Emerging memory technologies like phase change memory (PCM) [221], resistive RAM (ReRAM) [45], and spin-transfer torque RAM (STT-RAM) [19, 158, 223, 255] improve density compared to traditional DRAM while achieving varying degrees of performance parity. Concurrently, interconnect technologies like compute express link (CXL) [60] allow for technology-agnostic expansion of main memory capacity and direct access from accelerators like GPUs [22, 91]. The performance impact of these technologies on accelerator performance is of considerable importance but heavily understudied. This work aims to fill that gap, contributing to a broad category of solutions that attempt to transparently expand GPU memory capacity while hiding the associated performance costs.

In this chapter, we characterize the performance impact of heterogeneous host memory on accelerator performance using an Intel Optane and Nvidia A100 equipped system. We first present basic bandwidth measurements when moving data between host and GPU under different host memory configurations, showing how host to GPU and GPU to host bandwidth drop by up to 41% and 92%, respectively, compared to traditional DRAM memory. Prior work has shown that CXL-expanded memory only achieves up to 47% and 30% of the theoretical maximum bandwidth of the underlying DRAM memory [262], while highlighting significant performance variations across CXL controller architectures and the underlying memory technology.

To understand the real world impact of this performance deficit, we evaluate the inference performance of the open pre-trained transformer (OPT) family of LLMs [308] under various memory configurations using FlexGen [254], a state-of-the-art LLM serving framework that supports distribution of model weights across GPU memory, host memory, and permanent storage. Our results show an average 33% increase in per-layer processing time for OPT-175B with Optane as main memory compared to DRAM main memory, a direct result of the lower Optane bandwidth and the memory bound nature of LLM inference. While compression helps reduce this memory bottleneck, a deeper analysis of FlexGen's compute schedule reveals an imbalance in the compute/communication pipeline as the root cause. This imbalance is a byproduct of its weight placement scheme.

We address the imbalance with two alternate weight placement schemes, one each optimizing for latency and throughput. The first scheme, called HeLM, allocates weights for each layer in a compute time-aware fashion to improve the overlap of compute time of layer *i* with the weight transfer time of layer *i+1*. This allows HeLM to achieve a more balanced compute/communication pipeline, improving average time between tokens (TBT) by 27%. The second scheme, called All-CPU, offloads all weights to host memory and leaves GPU memory for key/value caches and hidden state. This boosts the maximum possible batch size from 8 to 44 and brings about a 5x improvement in throughput. HeLM's TBT and All-CPU's throughput come within 9% and 6% of an all-DRAM system, highlighting how careful data placement can help hide the performance deficiencies of emerging memory technologies.

In summary, this chapter makes the following contributions:

1. Quantification of host/device data movement performance with Intel Optane, a high capacity but low performance byte-addressable memory, as host memory, showing significantly lower bandwidth compared to traditional DRAM.

2. Characterization of LLM performance on a real system when using such memory, pointing at inefficient weight placement as a performance bottleneck.

3. Evaluation of two model weight placement schemes that optimize for latency and throughput, performing within 9% and 6% of an all-DRAM system, respectively.

4. Performance projections on to CXL-enabled memory, highlighting efficacy of the proposed policies across a range of memory performance characteristics.

5.1 Background

5.1.1 LLM Servers

Given the exponential growth in LLM model sizes, storing all model weights on accelerator memory has been increasingly challenging [87]. In response, several LLM frameworks have been proposed that enable offloading parts of the model to host memory or a backing store like disks. FlexGen [254] is one such framework that distributes model weights, KV cache, and hidden state between GPU memory, host main memory, and storage, and employs a ”zig-zag” compute schedule on the GPU that optimizes for throughput and weight reuse. The compute schedule overlaps the computation of a batch of requests on layer *j* (e.g., MHA) with the loading of layer *j+1* (e.g., FFN) and its associated KV cache onto the GPU. Listing 5.1 shows FlexGen’s computation schedule. Our characterization focuses on the overlap of compute with weight transfer given that the model weights are often at least an order of magnitude larger than both the KV cache and the hidden state.

Some LLM serving frameworks like llama.cpp [86], PowerInfer [258], and PowerInfer-2 [292] support concurrent CPU/GPU computation to avoid weight transfer bottlenecks. Among these servers, only PowerInfer-2 supports offloading to GPU memory, host memory, *and* storage. However, PowerInfer-2 is not open source. We therefore use FlexGen running on GPUs for our evaluation.

Listing 5.1: FlexGen computation schedule

```
for i in range(execute_gen_len):
    for j in range(num_layers):
        load_weight(i, j+1)
        compute_layer(i, j)
        sync()
```

llama.cpp [86] partitions model layers between the CPU and GPU, processing respective layers locally (model parallelism) and moving activations across whenever needed. LLM in a flash [10] improves inference latency on mobile devices by maximizing the reuse of weights loaded from flash memory into DRAM, and by optimizing weight layout in flash memory to improve access bandwidth. PowerInfer [258] exploits the power-law distribution of model weights, extracted from offline profiling, to distribute them across host and GPU memory. This allows the CPU and GPU to compute on sparse and dense matrices, respectively, in parallel. PowerInfer-2 [292] further optimizes inference for mobile devices by offloading dense and sparse computations on the NPU and CPUs, respectively, alongside optimizing I/O operations to enable streaming of weights from flash memory.

Among the presented servers, only FlexGen and PowerInfer-2 support offloading to GPU memory, host memory, *and* storage. However, PowerInfer-2 is not open source and we, therefore, use FlexGen for our evaluation.

5.1.2 Emerging Memory Technologies

Section 2.3 provides a detailed background on Intel Optane and Compute Express Link. This section summarizes some of their key features relevant to this work.

Intel Optane is a PCM-based byte-addressable non-volatile memory that offers significantly more density than traditional DRAM technology [121, 50]. While Optane fits into regular DDR4 slots and can be used accessed using regular loads/stores, it provides significantly lower performance. Prior work has shown Optane achieves nearly 2.5x lower sequential read bandwidth compared to DRAM and about 6x lower write bandwidth [129, 218, 293]. Being PCM-based also limits the life of each memory module in terms of its write endurance [121].

Compute Express Link (CXL) [60] was announced in 2019 [37] as an industry-standard interconnect technology to connect processors, devices, and memory expanders over PCIe bus interface. Of particular note is CXL's ability to allow for coherent expansion of main memory capacity over PCIe. By providing load/store semantics similar to traditional main memory, CXL memory provides transparent expansion of main memory without the limitations of traditional DDR interfaces. Moreover, the memory technology across the interconnect is not bound to be DRAM, allowing for use of high density media like SSDs [125, 140, 216, 243, 295] or even Optane itself [125]. This ensures broader applicability of findings presented in this chapter, especially in light of the discontinuation of Intel Optane [124].

While the performance of CXL memory is largely determined by the backing memory technology, communication over PCIe does provide an upper bound. CXL adds at least 70 nanoseconds to the round-trip latency [252], not accounting for contention at the expander. The achievable bandwidth is also limited to 64 GB/s for the latest PCIe 5.0 x16 link [284]. In comparison, our DDR4-based evaluation system achieves 157 GB/s across 8 memory channels (Section 5.2.1).

5.2 Evaluation Methodology

5.2.1 Platform

We perform our evaluation on an Intel Optane-equipped dual-socket machine. The configuration is listed in Table 5.1. Each socket has 4 memory controllers with 32 GB DDR4-2933 DRAM and 128 GB Optane DCPMM per channel/controller, providing a total of 256 GB DRAM and 1 TB Optane across the system. The system is paired with a Nvidia Ampere-based A100 GPU using 16 PCIe Gen 4 links that provide a maximum theoretical bandwidth of 32.0 GB/s. The GPU has 40 GB of HBM2 memory, organized as 5 stacks with 8 memory dies per stack [201].

Our evaluation considers all available configurations of Optane/DRAM. This includes Optane as storage with ext4-DAX file system [21], Optane Memory Mode (Optane main memory with DRAM cache), and Optane + DRAM main memory which is enabled by the Memkind library [30]. The last configuration exposes Optane memory as memory-only NUMA nodes.

5.2.2 Benchmarks

We use NVIDIA nvbandwidth [202] for basic bandwidth measurements between host and GPU. Our characterization presents results for both the NUMA nodes and with all combinations of Optane/DRAM host memory. To quantify real world performance impact of Optane on GPU performance, we use FlexGen [254] to run two variants

Table 5.1: System configuration

CPU	
Model	Dual socket Intel Xeon Gold 6330 (Ice Lake)
Frequency	Base: 2.0 GHz, turbo: 3.1 GHz
Cores (per socket)	28 (56 threads)
Memory (per socket)	4 memory controllers 16 GB DDR4-2933 DRAM x2 (per controller) 128 GB Optane (200 series) x1 (per controller)
GPU	
Model	Nvidia A100
Memory	40GB HBM2 (1215 MHz, 1555 GB/s)
Interface	PCIe Gen 4 x16 (32.0 GB/s)

of OPT models [308], OPT-30B and OPT-175B. OPT-30B models the scenario where model size surpasses GPU memory but fits in host DRAM. OPT-175B pushes further and surpasses host DRAM memory but fits in Optane memory, allowing us to compare the performance of heterogeneous main memory to traditional disk offloading. OPT-30B and OPT-175B contain 48 and 96 decoder blocks, resulting in 96 and 192 hidden layers (MHA + FFN), respectively. Along with one input embedding layer and one output embedding layer, the models have a total of 98 and 194 layers. Table 5.2 lists the various memory configurations we evaluate for each model. The input and output sequence lengths are limited to 128 and 21 tokens, respectively. We use prompts from the C4 dataset [222] and repeat each prompt 10 times.

5.2.3 Metrics

We evaluate LLM performance using three key metrics: time to first token (TTFT), time between tokens (TBT), and throughput in terms of tokens per second. TTFT

Table 5.2: LLM model/memory configuration

Model (# of Decoders)	Memory Configuration			Label
	SSD	Optane	DRAM	
OPT-30B (48)	N/A	N/A	Memory	DRAM
		Memory	N/A	NVDRAM
			Cache	MemoryMode
OPT-175B (96)	Storage	N/A	Memory	SSD
	N/A	Storage	Memory	FSDAX
		Memory	N/A	NVDRAM
			Cache	MemoryMode

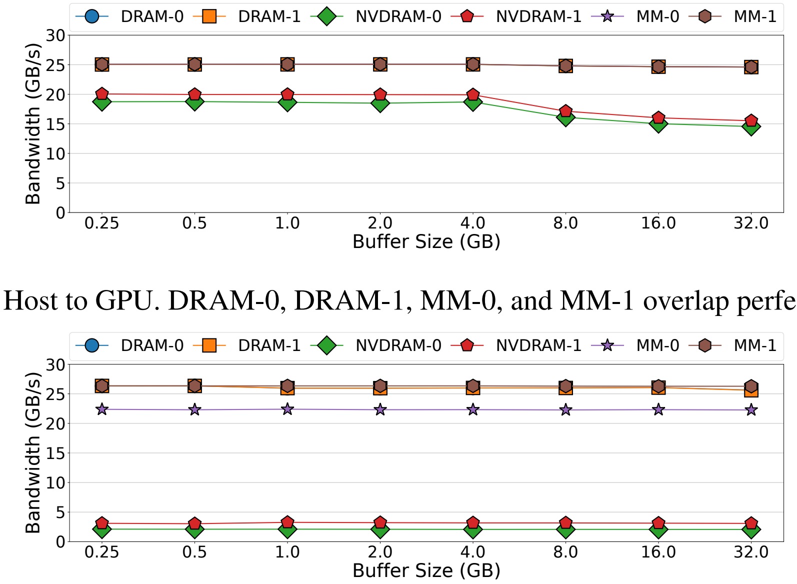
measures prefill latency, the inference stage that processes the entire input prompt. TBT measures decode latency, the successive stages of inference that utilize the KV cache from the prefill stage alongside the previously generated token to generate the next token. Finally, throughput measures the overall token generation rate across the entire process. For each metric, we present the arithmetic mean across all its values except the first, which we discard to account for cold start effects.

5.3 GPU/Host Data Movement Characterization

5.3.1 Basic Bandwidth Measurements

Figure 5.1 presents host to GPU (5.1a) and GPU to host (5.1b) bandwidth for buffer sizes between 256 MB and 32 GB. The figure presents the bandwidth when copying to/from DRAM, Optane DRAM (NVDRAM), and Optane Memory Mode (MM) for both the NUMA nodes.

Figure 5.1a shows how host to GPU bandwidth suffers a near constant loss of 20% with NVDRAM compared to DRAM up to a buffer size of 4 GB, with NVDRAM



(a) Host to GPU. DRAM-0, DRAM-1, MM-0, and MM-1 overlap perfectly.

(b) GPU to host. DRAM-0, DRAM-1, and MM-1 overlap perfectly.

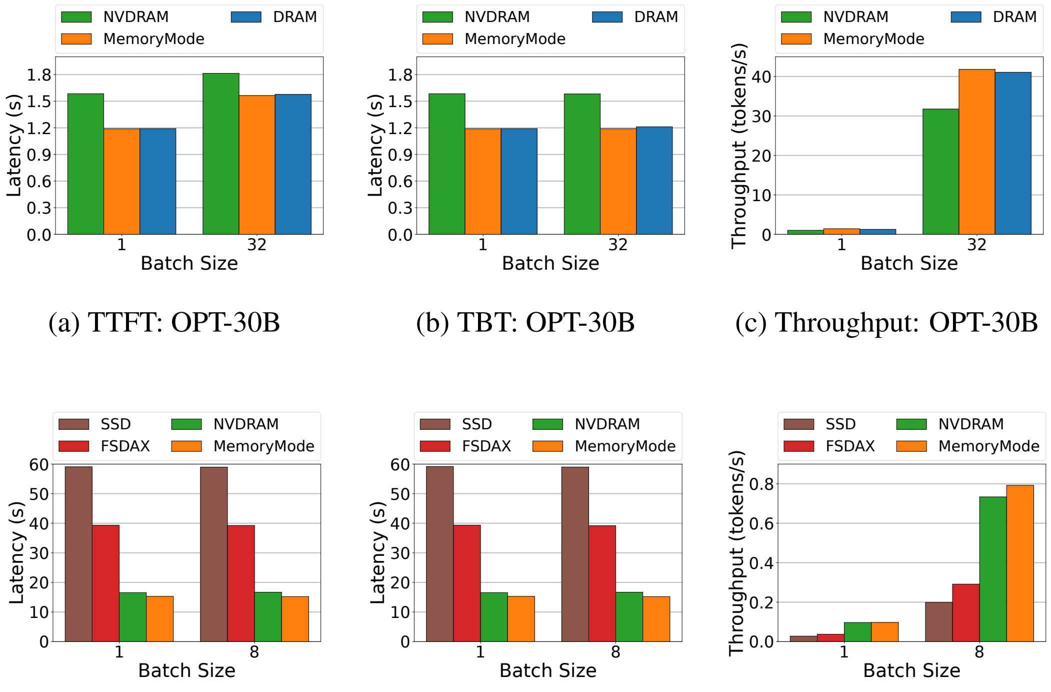
Figure 5.1: Host/GPU memory copy bandwidth. The numbers 0 and 1 represent the two NUMA nodes.

bandwidth dropping from 19.91 GB/s at 4 GB to 15.52 GB/s at 32 GB and increasing the performance deficit to 37%. We attribute this drop in performance to potentially non-consecutive data placement on NVM media due to wear-leveling and to misses in the Address Indirection Table (AIT) buffer that translates physical addresses to NVM media addresses [280, 293, 307]. MM is able to completely hide this performance gap, however, because the buffer size fits within the DRAM cache capacity (note that the MM and DRAM lines in Figure 5.1a overlap each other).

The performance gap between DRAM and NVDRAM is even wider when it comes to Optane's write performance. GPU to host bandwidth (Figure 5.1b) is 88% lower with NVDRAM compared to DRAM across all buffer sizes, maxing out at 3.26 GB/s with a buffer size of 1 GB. This bandwidth is consistent with prior observations [129]. We also notice how bandwidth for Optane is higher on NUMA node 1 compared to NUMA node 0. This is because the GPU is connected to PCIe ports local to node 1, meaning that accesses to node 0 need to go over the on-chip interconnect. Prior work has shown how Optane write performance worsens when accessed remotely [218]. Our own results using Intel Memory Latency Checker [274] also confirm this, including remote MM's inability to reach remote DRAM bandwidth.

5.3.2 LLM Performance

Figure 5.2 shows the performance of OPT-30B and OPT-175B models in terms of average TTFT, TBT, and throughput. The three metrics are presented for a batch size of 1 along with the maximum permissible size based on available GPU memory to avoid the KV cache impacting communication overheads (32 for OPT-30B and 8 for OPT-175B).



(a) TTFT: OPT-30B

(b) TBT: OPT-30B

(c) Throughput: OPT-30B

(d) TTFT: OPT-175B

(e) TBT: OPT-175B

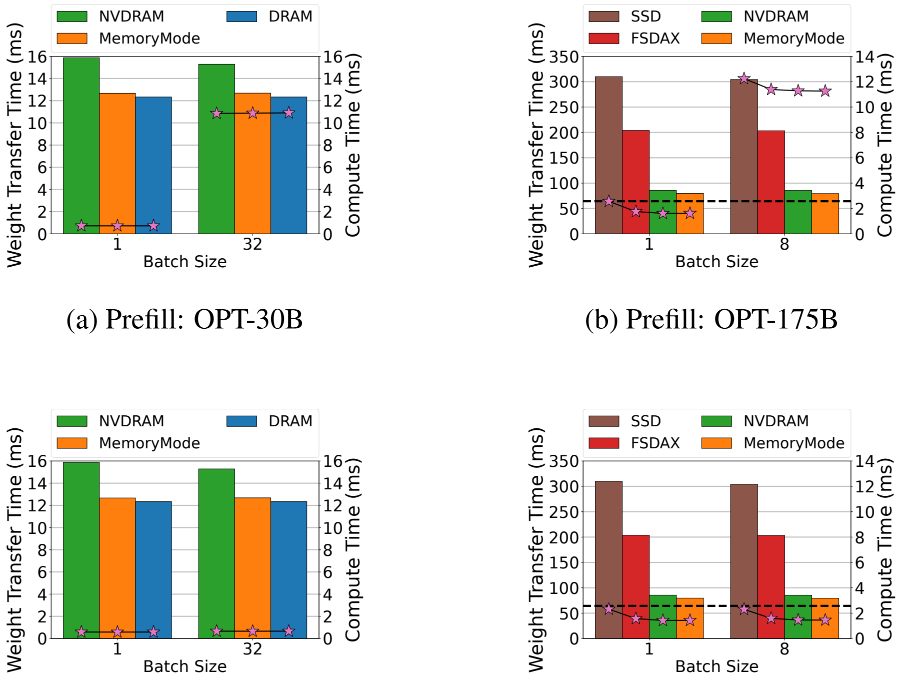
(f) Throughput: OPT-175B

Figure 5.2: Time to first token (TTFT), time between tokens (TBT), and throughput (tokens/s).

SSD and FSDAX configurations are, unsurprisingly, the slowest performing configurations. While FSDAX improves TTFT/TBT/throughput by 33.46%/33.48%/35.31% and 33.44%/33.58%/46.68% for OPT-175B with batch sizes 1 and 8, respectively, it falls short of reaching NVDRAM's performance. This is largely a result of Optane being exposed through the file system interface in FSDAX, requiring the use of a bounce buffer in DRAM when copying weights from Optane to GPU.

NVDRAM's lower bandwidth compared to DRAM impacts both TTFT and TBT significantly, hurting overall throughput. OPT-30B's TTFT increases by 33.03% and 15.05% with batch sizes 1 and 32, respectively, under NVDRAM compared to DRAM. Similarly, TBT goes up by 33.03% and 30.55% under the two batch sizes. This leads to a reduction in throughput by 18.96% and 22.68%, respectively. MemoryMode matches DRAM performance because the host-side model weights fit within DRAM cache. While there is no DRAM optima to compare against for OPT-175B, MemoryMode improves TTFT/TBT/throughput compared to NVDRAM by 7.67%/7.69%/0.60% and 8.90%/8.92%/7.98% for batch sizes 1 and 8, respectively. Keeping in mind that the model size outgrows the DRAM cache size here, an all-DRAM system likely performs even better than this.

Increasing batch sizes improves throughput almost linearly, as seen in Figures 5.2c and 5.2f. Ordinarily, prefill is compute bound because of its high operational intensity. As a consequence, TTFT tends to increase with an increase in batch size. We see this with OPT-30B where TTFT increases by 32.41%, 14.51%, and 31.50% under DRAM, NVDRAM, and MemoryMode configurations, respectively, when going from a batch size of 1 to 32 (Figure 5.2a). OPT-175B does not experience an increase in TTFT (Figure 5.2d) with increasing batch size because its large weight size makes its prefill stage memory bound. Decode, on the other hand, is memory bound because it consists of a series of GEMV computations which have low operational intensity. While increasing the batch size helps convert the GEMV computation in the feed forward network (FFN) stage (Section 2.5) into GEMM, each prompt must still perform a series of GEMV operations in the multi-head attention (MHA) stage (Section 2.5) with its own local KV cache. This limits the scaling of TBT with increasing batch sizes, as seen in Figures 5.2b and 5.2e.



(a) Prefill: OPT-30B

(b) Prefill: OPT-175B

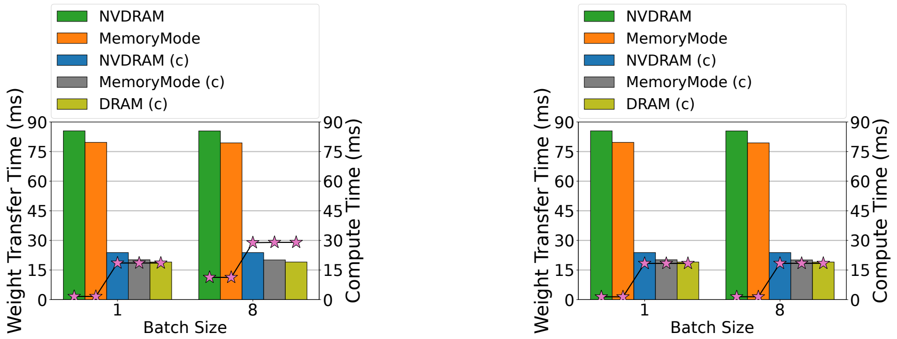
(c) Decode: OPT-30B

(d) Decode: OPT-175B

Figure 5.3: Compute/communication overlap during prefill and decode stages. The bars represent average weight transfer time while the line represents average compute time. The horizontal dashed line represents the ideal weight transfer time on an all-DRAM system. Note the different scales for the two y-axes in (b) and (d).

In order to better understand prefill and decode performance, we use FlexGen's built-in timers to get a breakdown of the time spent on compute and communication in each phase. Recall that FlexGen overlaps compute in layer *j* with the loading of weights for layer *j+1* (Section 5.1.1). Figure 5.3 presents this overlap on a per-layer basis for each model with different batch sizes, separate for both prefill and decode stages. Since both OPT-30B and OPT-175B consist of several decoder blocks (Table 5.2), the longer running operation within this pipeline affects the overall inference latency. For OPT-175B, we also measure and show the ideal average weight transfer time in an all-DRAM system by running the model with 8 decoder blocks instead of the default 96.

Figure 5.3 shows how the average weight transfer time under each memory configuration affects its TTFT and TBT performance (Figure 5.2), highlighting the memory-bound nature of LLM inference. Looking at OPT-30B's prefill stage (Figure 5.3a), we observe that the average compute time increases by about 15x for all three configurations when the batch size is increased from 1 to 32. This is the reason behind the increase in TTFT we observed earlier (Figure 5.2a) since many layers become compute-bound (even though the average compute time is below the average weight transfer time). The decode stage stays largely memory-bound, however, even with the large batch size (Figure 5.3c). Batching has been known to have minimal impact on the arithmetic intensity of the decode stage [213]. OPT-175B, meanwhile, is memory-bound in both prefill and decode stages because of its significantly larger weight sizes (hidden layer size of 12,288 versus OPT-30B's 7,168). While an all-DRAM system would improve the average weight transfer time in both stages by 32.78% and 22.41% compared to NVDRAM and MemoryMode, respectively, it will still be orders-of-magnitude higher than the compute time.



(a) OPT-175B: Prefill

(b) OPT-175B: Decode

Figure 5.4: Compute/communication overlap during prefill and decode stages with compression. The bars represent average weight transfer time while the line represents average compute time. The symbol *(c)* represents compressed configurations.

Compression/quantization is a well-known strategy to decrease model size, reducing both the model footprint and weight transfer time at the cost of increased computation, due to on-the-fly decompression, and potential accuracy loss. FlexGen supports compressing model weights down from FP16 to a 4-bit representation using group-wise quantization [253], reducing the model size to nearly a quarter with a negligible loss in accuracy [254]. This allows the model to fit entirely on host memory, even with traditional DRAM, obviating the need to offload to storage. Figure 5.4 highlights the compute/communication tradeoff of compression for OPT-175B for NVDRAM, MemoryMode, and DRAM configurations. Compared to the baseline, compression reduces weight transfer time by 72% and 74% for NVDRAM and MemoryMode, respectively, bringing it within 25% and 6% of DRAM ideal. The compute time, meanwhile, increases by anywhere between 2.5x-13x for both NVDRAM and MemoryMode configurations. Compression allows OPT-30B to fit fully into GPU memory, which we do not present.

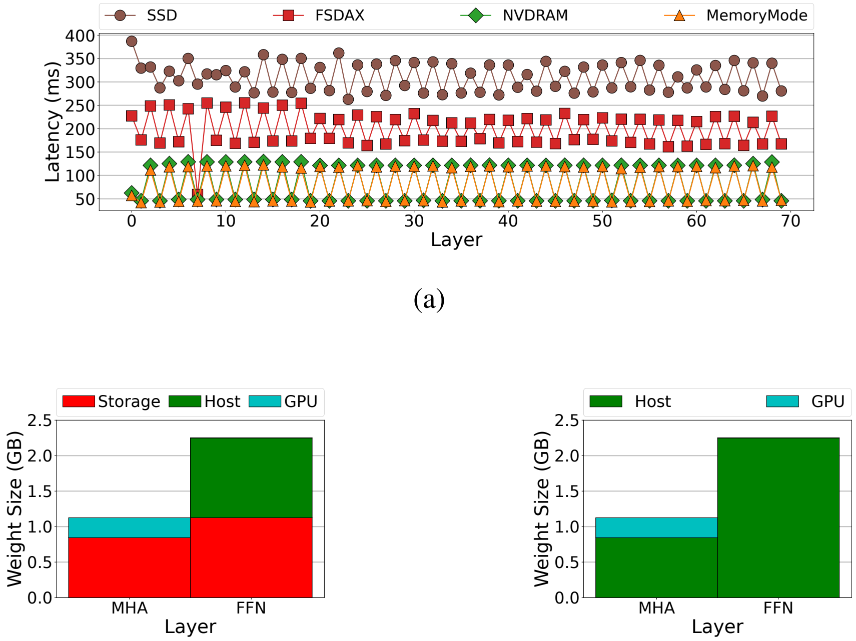
5.4 Impact of Weight Placement

Given the memory-bound nature of LLM inference, we take a closer look at the cost of transferring each weight to the GPU. We focus on model weight placement since the weight size dominates the total memory footprint. For instance, for a single OPT-175B self-attention block, the model weights occupy 3.38 GB of memory while the KV cache, the second highest contributor to the total memory footprint, occupies 47.98 MB for a batch size of 1 at the maximum context length of 2048 (72x smaller than weights). The total memory footprint of the model weights is 324.48 GB while that of the KV cache is 4.5 GB. For context, the GPU we use for our evaluation has 40 GB of onboard memory, which can hold the entire KV cache (4.5 GB), but not model weights (324.48 GB). We first evaluate the cost of transferring weights under FlexGen's existing weight strategy, highlighting an imbalance in compute and communication overlap. Based on our analysis, we propose two alternate weight placement schemes, both of which utilize compression to minimize data movement. The goal of the first scheme is to optimize for latency by mitigating the imbalance in compute and communication. The second scheme, meanwhile, optimizes for throughput by maximizing the batch size. We present these optimizations for NVDRAM and MemoryMode configurations only using OPT-175B, making a case for such memory technologies as an effective DRAM replacement for LLM inference.

5.4.1 FlexGen's Weight Transfer Costs

Figure 5.5a plots the latency of loading each layer of OPT-175B up to layer 70 of 194 for all memory configurations with compression. The plot has a striking saw-tooth pattern that continues all the way until layer 194 (not shown). This is a result of FlexGen's weight placement scheme, presented in Listing 5.2. Given a list of weights (`weight_specs`) and a user-specified percentage distribution across storage, host, and GPU (described in `policy`), the allocator, `init_weight_list`, distributes each layer's weights across the hierarchy to meet this goal. To achieve this, the function iterates over all the weights of the layer (line 17) and calculates the percentage contribution of the weights preceding weight *i* to the total layer size (lines 18-20). Based on this percentage and the input percentage distribution, `get_choice()` returns the device to allocate weight *i* on.

Our experiments show that this allocation scheme is imperfect and struggles to achieve the desired percentage distribution because differences in weight sizes do not lend themselves well to such a fine-grained distribution. For instance, for *(storage, host, GPU)* ratios of *(65, 15, 20)* under SSD/FSDAX configurations, the achieved overall weight distribution is *(58.6, 33.1, 8.3)*. Similarly, the input and achieved distribution for NVDRAM/MemoryMode is *(0, 80, 20)* and *(0, 91.7, 8.3)*, respectively. Furthermore, the weight distribution scheme is unaware of the relative size of each layer, which leads to the imbalanced weight transfer times across layers we see in Figure 5.5a. In



(a)

(b) SSD/FSDAX.

Storage: SSD/Optane, host: DRAM.

(c) NVDRAM/MemoryMode.

Host: Optane.

Figure 5.5: Per-layer weight load latency for a subset of OPT-175B layers (70/194) (a) and weight distribution of multi-head attention (MHA) and feed forward network (FFN) layers in SSD/FSDAX (b) and NVDRAM/MemoryMode (c) configurations.

particular, the dips and ridges in the figure correspond to multi-head attention (MHA) and feed forward network (FFN) layers, respectively. Figures 5.5b and 5.5c show the weight distribution of these two layers under SSD/FSDAX and NVDRAM/MemoryMode configurations, respectively. In both cases, we see how the larger FFN layer gets no allocation on the GPU while the smaller MHA layer does.

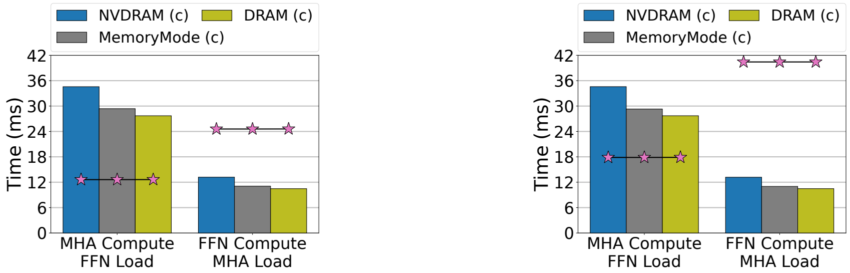
A direct consequence of this asymmetric weight distribution is that weigh transfer cannot be hidden effectively behind computation. Figure 5.6 shows the time spent in loading weights for FFN/MHA layers and how it overlaps with computing MHA/FFN for the prefill stage. MHA has a lower computation time than FFN, yet it is overlapped with the transfer of a larger set of weights because of FlexGen's weight distribution.

Listing 5.2: FlexGen weight allocation algorithm

```python
def get_device(cur_percent, percents, choices):
    percents = numpy.cumsum(percents)
    for i in range(len(percents)):
        if cur_percent < percents[i]:
            return choices[i]
    return choices[-1]

def init_weight_list(weight_specs, policy, env):
    dev_percents = [policy.disk_percent, policy.cpu_percent,
                    policy.gpu_percent]
    dev_choices = [env.disk, env.cpu, env.gpu]

    sizes = [spec.size for spec in weight_specs]
    sizes_cumsum = numpy.cumsum(sizes)

    for i in range(len(weight_specs)):
        mid_percent = (sizes_cumsum[i] - sizes[i] / 2) / \
                         sizes_cumsum[-1]
        dev = get_choice(mid_percent * 100, dev_percents,
                            dev_choices)
        dev.allocate(weight_specs[i])
```

5.4.2 HeLM: Latency Optimizing Weight Placement

In order to balance the compute/communication pipeline, we introduce **He***terogeneous* **L***ayerwise* **M***apping* (HeLM), a modified weight placement algorithm that attempts to equalize computation of layer *i* with weight transfer time of layer *i+1*. The key idea behind HeLM is to allocate more GPU space for layers whose transfer time will be overlapped with shorter computing layers. HeLM accomplishes this by allocating the weights of the first fully connected (FC) layer of FFN on the GPU, along with the weights of all the bias and normalization layers for both MHA and FFN. The rest of



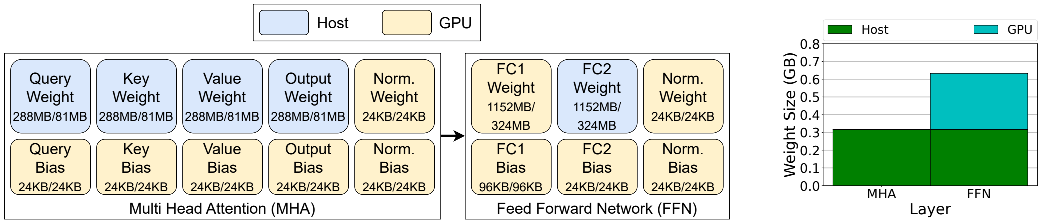
(a) Batch size 1

(b) Batch size 8

Figure 5.6: Overlap of MHA/FFN compute with the transfer of FFN/MHA weights in the prefill stage of OPT-175B with compression enabled. The bars represent average weight transfer time while the line represents average compute time. The overlap in decode stage with both batch sizes is nearly identical to prefill with batch size 1.

the MHA and FFN weights are offloaded on to the host memory. The algorithm is presented in Listing 5.3 and illustrated in Figure 5.7a.

Listing 5.3 shows how HeLM uses a custom weight distribution for MHA (lines 2-3) and FFN (lines 4-5) layers, along with sorting the weights in increasing order by size (line 13). Note that HeLM specifies device percentages in the order *(GPU, host, storage)*, instead of the default *(storage, host, GPU*.



(a)

(b)

Figure 5.7: (a) Breakdown of HeLM's weight distribution across host and GPU. The number under each weight is the uncompressed/compressed size of the weight. (b) HeLM's weight distribution.

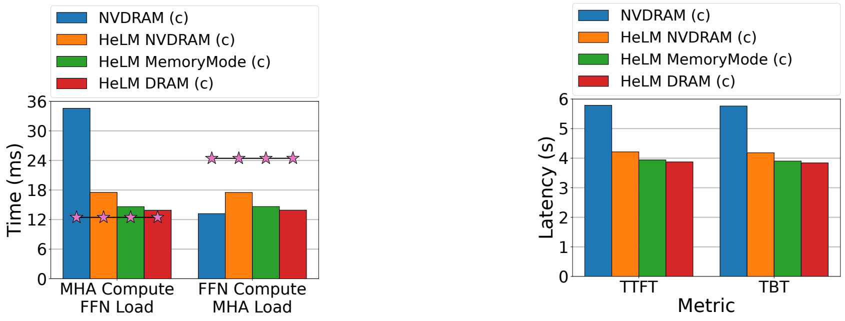
Figure 5.7b shows the weight distribution of MHA and FFN layers achieved by

Listing 5.3: HeLM weight allocation algorithm. The algorithm follows the default allocation algorithm line 14 onwards (Listing 5.2, line 13).

```
1  def init_weight_list(weight_specs, policy, env):
2      if is_mha(weight_specs):
3          dev_percents = [10, 90, 0]
4      elif is_ffn(weight_specs):
5          dev_percents = [30, 70, 0]
6      else:
7          dev_percents = [policy.gpu_percent, policy.cpu_percent,
8                          policy.disk_percent]
9  
10     dev_choices = [env.gpu, env.cpu, env.disk]
11 
12     weight_specs = list(sorted(weight_specs, key=lambda x: x.size)
13     sizes = [spec.size for spec in weight_specs]
14     ...
```

HeLM. This distribution reduces the time to transfer FFN weights by 49.33% while increasing it by 32.55% for MHA layers, as seen in Figure 5.8a. However, the increase in MHA load time is easily overlapped with FFN computation, leading to an overall reduction in layer processing time.

The balanced compute/communication pipeline directly results in improvements to inference latency. Figure 5.8b shows how HeLM improves TTFT and TBT on NVDRAM by 27.20% and 27.44% compared to the baseline scheme (Section 5.4.1). These numbers are within 8.75% and 8.91% of DRAM. MemoryMode, meanwhile, experiences an improvement of 31.90% and 32.28%, both of which are within 1.73% and 1.64% of DRAM. These results highlight how careful data placement can enable the use of Optane-like emerging memory technologies, and the heterogeneous configurations they enable, in latency-sensitive LLM serving scenarios.



(a) Compute/communication overlap

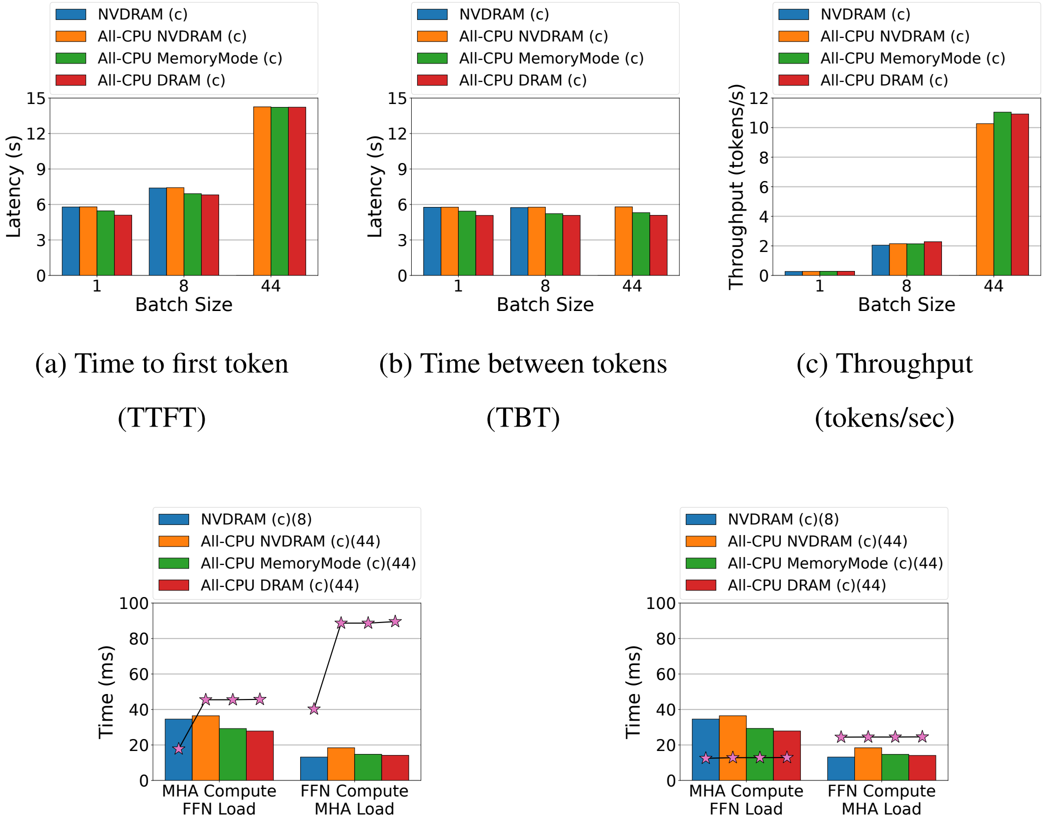
(b) TTFT and TBT

Figure 5.8: Impact of HeLM on (a) compute/communication overlap during decode and (b) time to first token (TTFT) and time between tokens (TBT). This is evaluated using OPT-175B with a batch size of 1. In Figure (a), the bars represent average weight transfer time while the line represents average compute time.

5.4.3 All-CPU: Throughput Optimizing Weight Placement

We study a second optimization called All-CPU where all weights are placed on host memory, leaving GPU memory for KV cache and hidden state. This makes sense because even with HeLM, only 33% of the total weights are held in the GPU memory. By pushing all of them out to host memory, we can trade weight transfer time for improved weight reuse enabled by a higher batch size. While this optimization has been explored before [254], we present it in the context of heterogeneous memory and evaluate how it fares compared to traditional DRAM.

Figures 5.9a, 5.9b, and 5.9c compare the TTFT, TBT, and throughput of All-CPU to the baseline scheme (Section 5.4.1) for batch sizes 1, 8, and 44, the latter of which is only possible with All-CPU. With all three batch sizes, the KV cache continues to fit inside GPU memory. All-CPU does not have a significant impact on either TTFT or TBT (1% degradation) or throughput (5% gain) with NVDRAM compared to the baseline at batch sizes 1 and 8. This highlights the minimal performance advantage of keeping model weights on GPU at all when optimizing for throughput. All-CPU makes



(a) Time to first token (TTFT)

(b) Time between tokens (TBT)

(c) Throughput (tokens/sec)

(d) Compute/communication overlap (prefill)

(e) Compute/communication overlap (decode)

Figure 5.9: Performance impact of All-CPU weight allocation on OPT-175B. Figures (d) and (e) compare compute/communication overlap with baseline weight allocation and batch size 8 to All-CPU and batch size 44. In both the figures, the bars represent average weight transfer time while the line represents average compute time.

better use of that space by allocating it to the KV cache instead and expanding the batch size. A key result here is the 5x increase in throughput when going from baseline NVDRAM at batch size 8 to All-CPU NVDRAM at batch size 44 (Figure 5.9c). In fact, the throughput at batch size 44 with All-CPU NVDRAM is within 6% of All-CPU DRAM.

Figures 5.9d and 5.9e show how the compute/communication overlap varies between the the baseline scheme with a batch size of 8 and All-CPU with a batch size of 44. While MHA weight transfer time increases significantly with All-CPU owing to FlexGen's weight allocation (Figure 5.5c), it is completely hidden behind computation in both prefill and decode stages. Interestingly, the compute time in decode stage does not increase when the batch size is increased from 8 to 44 (Figure 5.9e), indicating potential compute under-utilization. By maximizing the batch size, All-CPU improves compute utilization which leads to an overall increase in throughput (Figure 5.9c).

All-CPU MemoryMode reduces TTFT/TBT compared to All-CPU NVDRAM by 5.83%/5.77% with batch size 1, by 6.86%/9.46% with batch size 8, and by 0.24%/8.39% with batch size 44. It impacts the throughput the most at batch size 44, however, improving it by 7.57% and performing at-par with DRAM (1.15% better). This is evidence that optimized data placement on heterogeneous memory can not only achieve latency close to an all-DRAM system, but also throughput.

5.4.4 CXL Performance Projections

Like Intel Optane, CXL memory provides high capacity at the cost of performance. This cost varies based on both the CXL controller architecture as well as the underlying memory technology [262]. In order to evaluate the impact of our proposed optimizations on CXL memory, we borrow the bandwidth of two different CXL configurations from prior work and project the performance of each. These configurations are presented in Table 5.3. CXL-FPGA is based on evaluation presented by Sun et al. [262]

(called CXL-C in their paper) and uses an FPGA-based CXL controller backed by single channel DDR4-3200 memory. CXL-ASIC, meanwhile, is borrowed from Wang et al. [279] (called System A in their paper) and is based on an undisclosed commercial ASIC implementation backed by single channel DDR5-4800 memory. We utilize the bandwidth numbers for each configuration from its respective paper to project weight transfer times for each layer and calculate the achievable compute/communication overlap (Table 5.4), TTFT/TBT (Figure 5.10a), and throughput (Figure 5.10b), comparing it to NVDRAM.

Table 5.3: CXL configurations

Name	Memory Technology	Bandwidth (GB/s)
CXL-FPGA [262]	DDR4-3200 x1	5.12
CXL-ASIC [279]	DDR5-4800 x1	28

Table 5.4 shows the compute/communication overlap for each CXL configuration under all three weight allocation policies: baseline (Section 5.4.1), HeLM (Section 5.4.2), and All-CPU (Section 5.4.3). CXL-FPGA and CXL-ASIC cover a wide performance spectrum owing to differences in their CXL controller design. CXL-FPGA achieves considerably lower memory bandwidth than both NVDRAM and CXL-ASIC. The lower bandwidth means that CXL-FPGA stays largely memory bound across all weight allocation policies and inference stages, except All-CPU prefill with a batch size of 44. CXL-ASIC significantly outperforms both NVDRAM and CXL-FPGA, being the only configuration that achieves FFN load latency lower than MHA compute latency with HeLM. These results highlight how HeLM and All-CPU are able to improve the compute/communication overlap across a wide variety of CXL memory implementations.

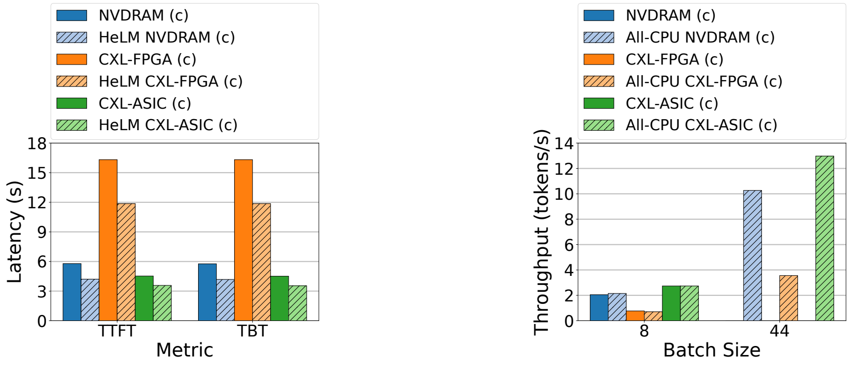
The improved compute/communication overlap with HeLM directly translates to lower inference latency, as shown in Figure 5.10a. HeLM improves TTFT/TBT by 27% and 21% for CXL-FPGA and CXL-ASIC, respectively. The improvements for

Table 5.4: Overlap of compute and communication with different weight allocation policies under NVDRAM configuration and the three different CXL configurations. A ratio of 1 indicates perfect overlap, while lower and higher values indicate memory-boundedness and compute-boundedness, respectively.

Allocation	Batch	Stage	MHA compute/FFN Load (ratio)			FFN Compute/MHA Load (ratio)		
Policy	Size		NVDRAM (c)	CXL-FPGA (c)	CXL-ASIC (c)	NVDRAM (c)	CXL-FPGA (c)	CXL-ASIC (c)
Baseline	1	Prefill	0.36	0.1	0.56	1.86	0.53	2.9
		Decode	0.36	0.1	0.55	1.85	0.53	2.88
	8	Prefill	0.52	0.14	0.79	3.07	0.87	4.77
		Decode	0.36	0.1	0.55	1.85	0.53	2.88
HeLM	1	Prefill	0.72	0.2	1.12	1.4	0.4	2.18
		Decode	0.71	0.2	1.1	1.4	0.4	2.16
All-CPU	1	Prefill	0.37	0.1	0.56	1.41	0.4	2.18
		Decode	0.36	0.1	0.55	1.39	0.39	2.16
	8	Prefill	0.51	0.14	0.79	2.3	0.65	3.57
		Decode	0.36	0.1	0.55	1.39	0.39	2.16
	44	Prefill	1.25	0.37	2.01	4.82	1.43	7.84
		Decode	0.35	0.1	0.57	1.33	0.4	2.16

CXL-ASIC come from reducing FFN weight transfer time and increasing MHA weight transfer time, thereby balancing compute with communication. CXL-FPGA, on the other hand, performs better simply because HeLM is able to store more weights on the GPU compared to the baseline scheme.

The gains from All-CPU are more varied. While NVDRAM and CXL-ASIC experience nearly the same performance with both baseline and All-CPU at batch size 8, CXL-FPGA suffers an 8.35% drop in throughput due to its poor memory performance. Nonetheless, both CXL-ASIC and CXL-FPGA achieve 4.74x and 5.04x higher throughput when going from the baseline scheme at batch size 8 to All-CPU at batch size 44, highlighting the efficacy of the placement scheme regardless of memory performance.



(a)

(b)

Figure 5.10: Projected performance improvements offered by HeLM (batch size=1) (a) and All-CPU (b) on CXL-based systems using OPT-175B.

5.5 Related Work

LLM memory optimizations: The scale of machine learning models is a long standing problem with several algorithmic and systems solutions. Pruning/sparsification [103, 159] and quantization [59, 118] are among the oldest and most well-known solutions. Pruning reduces model size by zero-ing out weights to reduce computation and memory footprint while aiming to preserve model accuracy, and has been shown to be effective for LLMs [180, 288]. Quantization compresses the model by using smaller bit-width representations for weights [94, 302], KV cache [117, 176], and/or activations [118] while also minimizing accuracy loss. This can be performed either during training (Quantization-Aware Training) [175, 253], where the model is retrained on quantized weights, or post-training (Post-Training Quantization) [289, 297], which generally scales better for large models. FlexGen adopts the second approach, compressing weights down to four bits using group-wise quantization (GWQ) [253]. GWQ divides weights into groups (64 in FlexGen), normalizes each value to the maximum value in its group and scaling it by the maximum supported value size (15 for 4 bits). This has shown to preserve accuracy for both encoder-only models like BERT [253] and decoder-only models like OPT [254].

KV cache is the second highest contributor to the total memory footprint of a LLM during inference, accounting for as much as 30% of the total size [154]. Unlike model parameters, KV cache grows dynamically with input batch size and is a key determinant of the model throughput. PagedAttention [154] applies ideas from operating system virtual memory management to manage this cache at a page-like block granularity to minimize memory waste and fragmentation, where each block contains KV values for a fixed number of tokens. S^3 [134] also minimizes KV cache size by predicting output sequence length for each prompt and limiting its associated cache size. CachedAttention [83] offloads inactive KV cache from GPU to host memory and disk during multi-turn conversations to prevent recomputation on older prompts. This cache is then proactively brought back to the GPU based on a predictor, along with overlapping communication with layer-wise computation. These approaches can be combined with our work to further increase batch sizes.

The memory-bound nature of LLMs makes them a great fit for processing in memory (PIM) architectures, as we saw in Chapter 4. In particular, LLMs can leverage host/PIM concurrency by overlapping query/key/value generation and self-attention on host and PIM, respectively [111, 212], or by distributing fully connected layers between host and PIM at a head granularity [248]. These architectures address the computational challenges of LLM inference but not capacity.

Application-specific optimizations for Optane: Use of Intel Optane DCPMM is of particular interest to HPC and ML community given the large memory footprint of their applications. Patil et al. [214] characterize the performance of HPC applications under App Direct and Memory Mode configurations, highlighting significant performance drops with former and the efficacy of the latter, especially when application working set sizes fit in the DRAM cache. Weiland et al. [282] demonstrate the performance scaling of materials simulation (CASTEP [56]), using Memory Mode to reduce execution time and energy consumption, and weather forecasting (IFS [182]), using App Direct mode to improve I/O performance, across a distributed system using Optane. Venkatesh et al. [273] characterize the performance of producer/consumer-based HPC workflows using Optane as intermediate storage, providing performance recommendations based on the access pattern of the producer and consumer. AutoTM [112] is a data management framework for machine learning training that solves a profile-guided integer linear program (ILP) to optimally place tensors in either DRAM or Optane, optimizing for execution time under a DRAM capacity constraint. Ren et al. [230] develop data placement and movement scheme for WarpX, an exascale plasma simulation tool. The proposed scheme optimizes for execution time by statically partitioning data between DRAM and PM, and performing dynamic data movement based on a performance model. Our work demonstrates how careful placement of LLM weights between Optane and GPU memory can compensate for Optane's slower performance compared to DRAM.

CXL-based tiered memory: TPP [186] employs a sampling and Linux LRU-based page hotness classification scheme to keep hot and cold pages in CXL and local memory, respectively, alongside asynchronous page allocation and reclamation. Pond [167] is a tiered CXL-based system for cloud vendors that uses a machine learning model to predict local vs. CXL allocation size for each virtual machine (VM), built on the observation that most VMs are insensitive to higher memory access latency. Pond also integrates a run time QoS monitoring system that handles mispredictions by allocating more local memory. Compared to these application-agnostic schemes, the presented work shows how application-specific optimizations can sufficiently hide the performance deficiencies of heterogeneous host memory. These insights are applicable to CXL too since CXL memory can be backed by SSDs [125, 140, 216, 243, 295] or even Optane itself [125].

5.5.1 GPU Memory Expansion

Our work very closely matches that of Choi et al. [52], where the authors evaluate the LLM serving performance of Nvidia Grace Hopper Superchip (GHS) [204], using vLLM framework [154] to serve LLaMa 3.1 8B, 70B, and 405B models [189]. There are several key differences between our work and theirs. First, GHS pairs only traditional DRAM with the CPU, while we evaluate the impact of emerging memory technologies and the heterogeneous configurations they enable. Second, unlike FlexGen, vLLM considers GPU memory as an inclusive cache. While this is similar to the All-CPU layout we evaluate, our work also evaluates a flat memory hierarchy where weight placement across GPU and host memory plays a key role in determining performance. Finally, GHS uses Nvidia NVLink [199] as the CPU/GPU interconnect which offers considerably higher bandwidth compared to the PCIe 4.0 interface used in ours [204], affecting the cost of CPU/GPU data movement. Keeping these differences in mind, we consider the two works to be complementary to each other.

Main memory: Zheng et al. [310] introduced the idea of “replayable far faults” wherein a GPU page fault appears as an ultra long latency memory access, avoiding stalling the SM and letting other warps execute. Combined with a prefetching scheme that saturates the PCIe bandwidth, they show a net performance improvement over the traditional “copy-then-execute” model. Nvidia’s Unified Memory (UM) [107] is largely based on this model and improves programmer productivity at the cost of performance [11, 79, 136]. This cost has spawned a large body of work analyzing and optimizing the prefetch/eviction policies [79, 80, 81, 89, 166], software hint-driven placement and prefetching [47], and GPU throttling and compression [166]. Alternatives to UM includes proposal like Demand MemCpy (DMC) [133], a hardware block added to the GPU memory management unit (GMMU) lazily copies data from host memory to GPU memory on-demand without interrupting the software. UVMMU [211] also extends GMMU to provide hardware support for page fault handling. Unlike DMC, however, UVMMU supports Unified Memory and, consequently, supports memory oversubscription. BW-AWARE [5, 6] is a heterogeneous memory page placement strategy for GPUs that divides the total allocation size across each memory type based on the ratio of its bandwidth to the total available system bandwidth. In addition, the authors evaluate a compiler-assisted scheme for applications with a non-uniform access pattern.

Storage: Allowing GPU direct access to storage is useful in minimizing data movement overheads for HPC and ML workloads. Nvidia GPUDirect Storage [265] facilitates copying of data directly from storage to GPU memory, avoiding CPU bounce buffers and overheads. NVMMU [306] integrates storage and GPU software stacks to reduce software overheads in GPU/storage transfers, alongside modifying to the storage driver to enable direct movement of data between GPU and storage without host bounce buffers. DRAGON [185] maps files from storage into Linux page cache and then exploits Unified Memory (UM) to transfer those pages to the GPU on-demand. G10 [305] improves ML training by mapping storage into UM page tables and moving tensors between storage, CPU, and GPU based on an offline tensor-liveness analysis stage. Pandey et al. [209] propose GPM, a library that enables fine-grained persistency support on byte-addressable non-volatile memory like Optane from GPUs. GPM provides a set of primitives that provide support for persistent transactions, logging, and checkpointing in GPU kernels. Based on this work, they propose a persistency model for GPUs [210].

CXL: Arif et al. [22] evaluate the impact CXL-enabled memory on GPU performance in a multi-GPU, multi-tenant scenario. The authors demonstrate the inefficacy of allocating application memory in a local DRAM-first fashion as well as of distributing DRAM memory uniformly across all GPUs, making a case for proportionally dividing DRAM capacity across all GPU kernels based on their total memory need. Gouk et al. [91] develop and synthesize a CXL controller for GPUs to directly access CXL Type-3 memory. Optimized for latency and evaluated with the open-source RISC-V based Vortex GPU [268], the controller integrates additional optimizations to prefetch loads and buffer writes to improve performance when interfacing with SSD-backed memory [125, 216, 243].

On-chip heterogeneous memory: Wang et al. [275] quantify the impact of on-chip PCM in GPUs, showing how it hurts both performance and energy efficiency. To rectify this, they propose a compiler-directed data placement scheme in a hybrid DRAM/PCM architecture along with hardware support for dynamic data movement between the two memories. Hong et al. [115] perform a design space exploration for storage-class memory backed DRAM cache architectures, presenting techniques to minimize tag probe overhead and cache bypassing to improve utilization. Prior work has also explored persistence on GPUs. Chen et al [43, 44] propose the use of "helper warps" in GPU kernels to move persistence off the application critical path. These warps can be adaptively turned off under high memory contention to reduce memory pressure. Lin et al. [171] propose an epoch persistency model for GPUs that can have varying scopes corresponding to GPU thread hierarchy.

5.6 Conclusion

As large language models continue to evolve, the growth in model sizes will continue to stress the memory subsystem for performance and capacity. This chapter shows how replacing DRAM with emerging technologies like Intel Optane can enable larger model sizes that fit in main memory, but not without a performance penalty. Diving deeper into the performance characteristics of running inference on OPT-30B and OPT-175B models with FlexGen, a LLM serving framework, we show how this performance degradation is largely a function of data placement and balancing computation with communication. We evaluate two alternate data placement schemes, one each optimizing for latency and throughput. The latency optimizing scheme, called HeLM, performs compute-time aware data placement that attempts to equalize the compute time of layer *i* and the weight transfer time of layer *i+1*. HeLM improves compute/communication pipeline balance and achieves token generation latency on Optane main memory within

9% of an all-DRAM system. The throughput optimizing scheme, called All-CPU, offloads all weights to the host memory, bumping the maximum possible batch size up from 8 to 44. The increased batch size helps All-CPU Optane net a throughput increase of 5x compared to baseline DRAM at a batch size of 8, while maintaining the same time between tokens. All-CPU Optane is within 6% of All-CPU DRAM, paving the way for models that exceed the capacity of DRAM. Our projections on CXL-enabled memory indicate that these findings remain valid for a broad spectrum of CXL devices. The presented techniques may be generalized to other models and frameworks by adapting to their compute schedule and data movement costs. We hope that the insights presented in this chapter inform the design of improved weight placement algorithms that can automatically make latency/throughput tradeoffs based on desired quality of service requirements.

6 Conclusion

Data movement bottlenecks have long limited the efficiency of computer systems and will continue to do so in the foreseeable future. This dissertation identified three critical data movement bottlenecks in accelerator-rich systems and proposed solutions that mitigate them in a cost effective manner.

First, Chapter 3 pinpoints inter-accelerator communication as a key issue that hampers both the performance and energy efficiency of mobile systems-on-chip (SoCs) that integrate multiple loosely-coupled accelerators. This dissertation proposes RELIEF, an online accelerator scheduling policy that maximizes the utilization of existing inter-accelerator communication hardware. RELIEF exploits the slack time of one application to prioritize requests from another application such that the latter's producer and consumer requests can be scheduled in consecutive order and, thus, communicate using specialized hardware. By being aware of each application's quality of service (QoS) requirements, RELIEF improves both energy consumption *and* fairness by 18% and 14%, respectively. The presented implementation of RELIEF achieves sub-microsecond average scheduling latency on a microcontroller and integrates into existing SoCs with little to no hardware modifications.

Second, Chapter 4 quantifies the negative performance impact of integrating processing in memory (PIM) into contemporary GPUs, especially in terms of fairness between concurrently executing PIM and non-PIM applications. This dissertation remedies these issues by proposing changes to the interconnect and memory controller, the two key points of contention for PIM and non-PIM requests. At the interconnect, separate virtual channels for the two request types prevents memory intensive PIM requests from degrading the flow of non-PIM requests. This ensures the memory controller has sufficient visibility into both the request streams. Such improved visibility helps the proposed memory controller scheduling policy, F3FS, to optimize for both fairness between the two request types and memory throughput by balancing locality for each request type and the frequency of switches between them. Compared to a baseline GPU-PIM system, the proposed architecture improves fairness and throughput by up to 72% and 22%, respectively. These modifications add minimal area and power overheads to existing GPUs.

Hardware support for accommodating PIM-enabled memory that requires switching between PIM and non-PIM modes [156, 163] is going to be key to the adoption of such technologies. The presented work is one of few recent proposals that consider the challenges involved therein [97, 162].

Finally, Chapter 5 characterizes the performance impact of emerging memory technologies like Intel Optane and CXL-enabled memory on the performance of a GPU serving an outsized large language model (LLM) that exceeds the capacity of both GPU and traditional DRAM-based host memory. Using a real Optane-based machine, the results highlight the remarkable impact data placement has on LLM inference performance. In response, this dissertation evaluates the performance of two alternative data placement strategies called HeLM and All-CPU. HeLM optimizes for inference latency by allocating weights in a layer compute time-aware fashion that improves the overlap of computation time with weight transfer time. All-CPU, meanwhile, optimizes for throughput by offloading all model weights to the CPU and utilizing GPU memory for key/value cache only, allowing for significantly larger batch sizes. Compared to a baseline compression-only scheme, the latency and throughput of HeLM and All-CPU on Intel Optane come within 9% and 6% of an all-DRAM ideal.

6.1 Future Work

The work presented in this dissertation can be expanded upon in many significant and impactful ways.

Dynamic locality-aware computation placement: RELIEF (Chapter 3) makes two simplifying assumptions: each computational kernel maps to a unique accelerator type, and that there is a single instance of each accelerator type. This sidesteps the issue of deciding which accelerator to place a kernel on. Prior solutions to this problem are either locality-oblivious [303] or application-specific [63]. RELIEF can potentially address both limitations since it already maintains metadata for application QoS requirements, accelerator execution times, and location of each accelerator's input data. An effective computation placement algorithm will need to consider the queuing delay, computation time, and data transfer time for each candidate accelerator. The complexity of choosing an accelerator, however, is non-trivial since it will affect the laxity of each following kernel in the computation DAG.

Distributed accelerator scheduling: As SoCs become more heterogeneous and incorporate an increasing number of accelerators, a centralized hardware manager will no longer be scalable. An ideal solution here would be for each accelerator to make a local scheduling decision that contributes to a globally optimal schedule while minimizing metadata communication between accelerators. A potential middle ground between purely centralized and distributed scheduling architectures could be a hierarchical solution that integrates several hardware managers. Each hardware manager would schedule operations on a subset of accelerators that communicate frequently, similar to prior work [152], and communicate with other managers only when an application chain crosses tile boundaries.

PIM-aware adaptive memory subsystem: The changes to PIM-enabled architectures proposed in this dissertation utilize several constants and fixed limits, like the fixed partitioning of PIM and non-PIM requests in the interconnect and the length of each PIM/non-PIM batch at the memory controller. While this improves average performance, it leaves the door open for techniques that can adapt these limits according to the access pattern of the executing applications. Such a design can help improve performance in the edge scenarios where the proposed policy is not the best performing (Figure 4.13).

Parallel PIM and non-PIM servicing: The PIM architectures evaluated in this dissertation switch modes at the granularity of a memory rank, even if all the banks in the rank cannot perform computation (Section 4.1.1). This resource wastage can be mitigated with architectural improvements that allow for the broadcast of PIM commands to a subset of banks (e.g., even/odd banks, bank groups, etc.) with parallel servicing of load/store requests in the remaining banks. While such an enhancement could make the PIM architecture more flexible and improve throughput, it will make the mode switching decision more granular and add a new dimension to memory controller scheduling that will need to account for parallel PIM/non-PIM servicing.

Runtime dynamic data placement and movement: Chapter 5 demonstrated the efficacy of two intelligent LLM data placement schemes, one each optimizing for latency and throughput. An algorithm that can utilize these schemes to traverse the latency/throughput spectrum based on desired QoS requirements would be the next logical step here. A key enabler for such an algorithm would be the ability to divide and manage smaller chunks of the weights of a single layer (e.g., fully connected), allowing for finer grained distribution of each layer across the memory hierarchy.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. event-place: Savannah, GA, USA.

[2] B. Abali, B. Blaner, J. Reilly, M. Klein, A. Mishra, C. B. Agricola, B. Sendir, A. Buyuktosunoglu, C. Jacobi, W. J. Starke, H. Myneni, and C. Wang. Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, Valencia, Spain, May 2020. IEEE. ISBN 978-1-72814-661-4. doi: 10.1109/ISCA45697.2020.00012. URL https://ieeexplore.ieee.org/document/9138913/.

[3] S. Aga, N. Jayasena, and M. Ignatowski. Co-ML: A Case for Collaborative ML Acceleration Using near-Data Processing. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, pages 506–517, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-7206-0. doi: 10.1145/3357526.3357532. URL https://doi.org/10.1145/3357526.3357532. event-place: Washington, District of Columbia, USA.

[4] N. Agarwal and T. F. Wenisch. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 631–644, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037706. URL https://doi.org/10.1145/3037697.3037706. event-place: Xi'an, China.

[5] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch. Unlocking bandwidth for GPUs in CC-NUMA systems. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 354–365, 2015. doi: 10.1109/HPCA.2015.7056046.

[6] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 607–618, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694381. URL https://doi.org/10.1145/2694344.2694381. event-place: Istanbul, Turkey.

[7] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 336–348, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-3402-0. doi: 10.1145/2749469.2750385. URL https://doi.org/10.1145/2749469.2750385. event-place: Portland, Oregon.

[8] J. Ajanovic. PCI Express (PCIe) 3.0 Accelerator Features. White paper, Aug. 2008. Intel.

[9] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong, D. Kim, W.-m. Hwu, and N. S. Kim. Application-Transparent Near-Memory Processing Architecture with Memory Channel Network. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 802–814, 2018. doi: 10.1109/MICRO.2018.00070.

[10] K. Alizadeh, S. I. Mirzadeh, D. Belenko, S. Khatamifard, M. Cho, C. C. Del Mundo, M. Rastegari, and M. Farajtabar. LLM in a flash: Efficient Large Language Model Inference with Limited Memory. In L.-W. Ku, A. Martins, and V. Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12562–12584, Bangkok, Thailand, Aug. 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.678. URL https://aclanthology.org/2024.acl-long.678/.

[11] T. Allen and R. Ge. In-depth analyses of unified virtual memory system for GPU accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8442-1. doi: 10.1145/3458817.3480855. URL https://doi.org/10.1145/3458817.3480855. event-place: St. Louis, Missouri.

[12] A. Amarnath, S. Pal, H. T. Kassa, A. Vega, A. Buyuktosunoglu, H. Franke, J.-D. Wellman, R. Dreslinski, and P. Bose. Heterogeneity-Aware Scheduling on SoCs for Autonomous Vehicles. *IEEE Computer Architecture Letters*, 20(2):82–85, 2021. doi: 10.1109/LCA.2021.3085505.

[13] AMD. AMD ROCm HIP Documentation: Stream Management, . URL https://rocm.docs.amd.com/projects/HIP/en/latest/doxygen/html/group___stream.html.

[14] AMD. AMD Vitis Unified Software Platform, . URL https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis.html.

[15] AMD. AMD Zynq UltraScale+ MPSoCs, . URL https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-mpsoc.html.

[16] AMD. AMD ROCm HIP Documentation: Memory Management, . URL https://rocm.docs.amd.com/projects/HIP/en/latest/doxygen/html/group___memory.html.

[17] AMD. AXI4-Stream Infrastructure IP Suite v3.0, May 2023. URL https://docs.xilinx.com/r/en-US/pg085-axi4stream-infrastructure/AXI4-Stream-Infrastructure-IP-Suite-v3.0-LogiCORE-IP-Product-Guide.

[18] S. Angizi and D. Fan. GraphiDe: A Graph Processing Accelerator leveraging In-DRAM-Computing. In *Proceedings of the 2019 Great Lakes Symposium on VLSI*, GLSVLSI '19, pages 45–50, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6252-8. doi: 10.1145/3299874.3317984. URL https://doi.org/10.1145/3299874.3317984. event-place: Tysons Corner, VA, USA.

[19] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi. Spin-transfer torque magnetic random access memory (STT-MRAM). *J. Emerg. Technol. Comput. Syst.*, 9(2), May 2013. ISSN 1550-4832. doi: 10.

1145/2463585.2463589. URL https://doi.org/10.1145/2463585.2463589. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[20] J. Archer. What is Resizable BAR, and should you use it?, Jan. 2025. URL https://www.rockpapershotgun.com/what-is-resizable-bar-and-should-you-use-it.

[21] L. K. Archives. Direct Access for files. URL https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[22] M. Arif, A. Maurya, and M. M. Rafique. Accelerating Performance of GPU-based Workloads Using CXL. In *Proceedings of the 13th Workshop on AI and Scientific Computing at Scale Using Flexible Computing*, FlexScience '23, pages 27–31, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701665. doi: 10.1145/3589013.3596678. URL https://doi.org/10.1145/3589013.3596678. event-place: Orlando, FL, USA.

[23] ARM. ARM Cortex-A7. URL https://developer.arm.com/Processors/Cortex-A7.

[24] ARM. AMBA AXI-Stream Protocol Specification, Apr. 2021. URL https://developer.arm.com/documentation/ihi0051/latest.

[25] K. Asanović and D. Patterson. Instruction Sets Should Be Free: The Case For RISC-V. Technical Report UCB/EECS-2014-146, Aug. 2014. URL https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf.

[26] M. Asri, C. Dunham, R. Rusitoru, A. Gerstlauer, and J. Beard. The Non-Uniform Compute Device (NUCD) Architecture for Lightweight Accelerator Offload. In *2020 28th Euromicro International Conference on Parallel,*

Distributed and Network-Based Processing (PDP), pages 38–45, 2020. doi: 10.1109/PDP50117.2020.00013.

[27] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–427, 2012. doi: 10.1109/ISCA.2012.6237036.

[28] S. Baruah. Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. In *25th IEEE International Real-Time Systems Symposium*, pages 37–46, 2004. doi: 10.1109/REAL.2004.20.

[29] S. K. Baruah. Partitioning real-time tasks among heterogeneous multiprocessors. In *International Conference on Parallel Processing, 2004. ICPP 2004.*, pages 467–474 vol.1, 2004. doi: 10.1109/ICPP.2004.1327956.

[30] M. Biesek. Memkind Support For KMEM DAX Option, Jan. 2020. URL https://pmem.io/blog/2020/01/memkind-support-for-kmem-dax-option/.

[31] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0-89791-700-6. doi: 10.1145/209936.209958. URL https://doi.org/10.1145/209936.209958. event-place: Santa Barbara, California, USA.

[32] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu. LazyPIM: An Efficient Cache Coherence Mechanism

for Processing-in-Memory. *IEEE Computer Architecture Letters*, 16(1):46–50, 2017. doi: 10.1109/LCA.2016.2577557.

[33] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 316–331, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173177. URL https://doi.org/10.1145/3173162.3173177. event-place: Williamsburg, VA, USA.

[34] J. Brown, S. Woodward, B. Bass, and C. Johnson. IBM Power Edge of Network Processor: A Wire-Speed System on a Chip. *IEEE Micro*, 31(2):76–85, 2011. doi: 10.1109/MM.2011.3.

[35] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Models are Few-Shot Learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[36] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang.

Sparks of Artificial General Intelligence: Early experiments with GPT-4, 2023. URL https://arxiv.org/abs/2303.12712. arXiv: 2303.12712.

[37] W. Calvert. Intel, Google and others join forces for CXL interconnect, Mar. 2019. URL https://www.datacenterdynamics.com/en/news/intel-google-and-others-join-forces-cxl-interconnect/.

[38] J. Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986. doi: 10.1109/TPAMI.1986.4767851.

[39] S. Charagulla. High-Bandwidth Memory Ready for AI Prime Time: HBM2e vs. GDDR6, May 2019. URL https://www.eeweb.com/high-bandwidth-memory-ready-for-ai-prime-time-hbm2e-vs-gddr6/.

[40] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. doi: 10.1109/IISWC.2009.5306797.

[41] Q. Chen, H. Yang, J. Mars, and L. Tang. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 681–696, Atlanta, Georgia, USA, 2016. Association for Computing Machinery. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872368. URL https://doi.org/10.1145/2872362.2872368.

[42] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization

in Warehouse-Scale Computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 17–32, Xi'an, China, 2017. Association for Computing Machinery. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037700. URL https://doi.org/10.1145/3037697.3037700.

[43] S. Chen, F. Zhang, L. Liu, and L. Peng. Efficient GPU NVRAM Persistence with Helper Warps. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6725-7. doi: 10.1145/3316781.3317810. URL https://doi.org/10.1145/3316781.3317810. event-place: Las Vegas, NV, USA.

[44] S. Chen, L. Liu, W. Zhang, and L. Peng. Architectural Support for NVRAM Persistence in GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 31(5):1107–1120, 2020. doi: 10.1109/TPDS.2019.2960233.

[45] Y. Chen. ReRAM: History, Status, and Future. *IEEE Transactions on Electron Devices*, 67(4):1420–1433, 2020. doi: 10.1109/TED.2019.2961505.

[46] Y.-T. Chen, J. Cong, M. A. Ghodrat, M. Huang, C. Liu, B. Xiao, and Y. Zou. Accelerator-rich CMPs: From concept to real hardware. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 169–176, 2013. doi: 10.1109/ICCD.2013.6657039.

[47] S. Chien, I. Peng, and S. Markidis. Performance Evaluation of Advanced Features in CUDA Unified Memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 50–57, 2019. doi: 10.1109/MCHPC49590.2019.00014.

[48] B. Y. Cho, Y. Kwon, S. Lym, and M. Erez. Near Data Acceleration with Concurrent Host Access. In *2020 ACM/IEEE 47th Annual International Symposium on*

Computer Architecture (ISCA), pages 818–831, 2020. doi: 10.1109/ISCA45697.2020.00072.

[49] K. Cho, B. v. Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, 2014. arXiv: 1406.1078.

[50] J. Choe. Intel's 2nd Generation XPoint Memory - Will it be worth the long wait ahead?, Apr. 2021. URL https://www.techinsights.com/blog/memory/intels-2nd-generation-xpoint-memory.

[51] J. Choi, H. Y. Yeom, and Y. Kim. Implementing CUDA Unified Memory in the PyTorch Framework. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pages 20–25, 2021. doi: 10.1109/ACSOS-C52956.2021.00029.

[52] W. Choi, J. Jeong, H. Jang, and J. Ahn. GPU-Centric Memory Tiering for LLM Serving With NVIDIA Grace Hopper Superchip. *IEEE Computer Architecture Letters*, 24(1):33–36, 2025. doi: 10.1109/LCA.2025.3533588.

[53] Y. Choi and M. Rhu. PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 220–233, 2020. doi: 10.1109/HPCA47549.2020.00027.

[54] L. Chua. Memristor-The missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, 1971. doi: 10.1109/TCT.1971.1083337.

[55] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 30–40, 2004. doi: 10.1109/MICRO.2004.5.

[56] S. J. Clark, M. D. Segall, C. J. Pickard, P. J. Hasnip, M. I. J. Probert, K. Refson, and M. C. Payne. First principles methods using CASTEP. *Zeitschrift für Kristallographie - Crystalline Materials*, 220(5-6):567–570, 2005. doi: doi:10.1524/zkri.220.5.567.65075. URL https://doi.org/10.1524/zkri.220.5.567.65075.

[57] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman. Architecture Support for Domain-Specific Accelerator-Rich CMPs. *ACM Trans. Embed. Comput. Syst.*, 13(4s), Apr. 2014. ISSN 1539-9087. doi: 10.1145/2584664. URL https://doi.org/10.1145/2584664. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[58] E. G. Cota, P. Mantovani, G. D. Guglielmo, and L. P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015. doi: 10.1145/2744769.2744794.

[59] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: training deep neural networks with binary weights during propagations. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pages 3123–3131, Cambridge, MA, USA, 2015. MIT Press. event-place: Montreal, Canada.

[60] CXL Consortium. Compute Express Link. URL https://computeexpresslink.org/.

[61] CXL Consortium. Compute Express Link 1.1 Specification, June 2019. URL https://computeexpresslink.org/.

[62] CXL Consortium. Compute Express Link 3.0 Specification, Aug. 2022. URL https://computeexpresslink.org/.

[63] I. Dagli and M. E. Belviranli. Shared Memory-contention-aware Concurrent DNN Execution for Diversely Heterogeneous System-on-Chips. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '24, pages 243–256, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704352. doi: 10.1145/3627535.3638502. URL https://doi.org/10.1145/3627535.3638502. event-place: , Edinburgh, United Kingdom,.

[64] W. J. Dally, S. W. Keckler, and D. B. Kirk. Evolution of the Graphics Processing Unit (GPU). *IEEE Micro*, 41(6):42–51, 2021. doi: 10.1109/MM.2021.3113475.

[65] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. doi: 10.1109/JSSC.1974.1050511.

[66] M. L. Dertouzos and A. K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 1989. doi: 10.1109/32.58762.

[67] M. Ditty, A. Karandikar, and D. Reed. Nvidia's Xavier SoC. Cupertino, California, USA, Aug. 2018. URL https://old.hotchips.org/hc30/1conf/1.12_Nvidia_XavierHotchips2018Final_814.pdf.

[68] M. Du and M. L. Scott. Buffered Persistence in B+ Trees. *Proc. ACM Manag. Data*, 2(6), Dec. 2024. doi: 10.1145/3698801. URL https://doi.org/10.1145/3698801. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[69] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *2003 12th International Conference on*

Parallel Architectures and Compilation Techniques, pages 220–231, 2003. doi: 10.1109/PACT.2003.1238018.

[70] Y. Eckert, N. Jayasena, and G. H. Loh. Thermal Feasibility of Die-Stacked Processing in Memory. In *2nd Workshop on Near-Data Processing*, Cambridge, UK, Dec. 2014.

[71] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 978-1-4503-0472-6. doi: 10.1145/2000064.2000108. URL https://doi.org/10.1145/2000064.2000108. event-place: San Jose, California, USA.

[72] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008. doi: 10.1109/MM.2008.44.

[73] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 283–295, 2015. doi: 10.1109/HPCA.2015.7056040.

[74] K. Fatahalian. The Rise of Mobile Visual Computing Systems. *IEEE Pervasive Computing*, 15(2):8–13, Apr. 2016. ISSN 1536-1268. doi: 10.1109/MPRV.2016.35. URL http://ieeexplore.ieee.org/document/7445776/.

[75] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48,

New York, NY, USA, 2012. Association for Computing Machinery. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2150982. URL https://doi.org/10.1145/2150976.2150982. event-place: London, England, UK.

[76] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development*, 54(1):3:1–3:11, 2010. doi: 10.1147/JRD.2009.2036980.

[77] N. Friedman. Introducing GitHub Copilot: your AI pair programmer, June 2021. URL https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/.

[78] Y. Gan, Y. Qiu, L. Chen, J. Leng, and Y. Zhu. Low-Latency Proactive Continuous Vision. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, pages 329–342, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-8075-1. doi: 10.1145/3410463.3414650. URL https://doi.org/10.1145/3410463.3414650. event-place: Virtual Event, GA, USA.

[79] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem. Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 224–235, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6669-4. doi: 10.1145/3307650.3322224. URL https://doi.org/10.1145/3307650.3322224. event-place: Phoenix, Arizona.

[80] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem. Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscrip-

tion. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–461, 2020. doi: 10.1109/IPDPS47924.2020.00054.

[81] D. Ganguly, R. Melhem, and J. Yang. An Adaptive Framework for Oversubscription Management in CPU-GPU Unified Memory. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1212–1217, 2021. doi: 10.23919/DATE51398.2021.9473982.

[82] K. K. Ganju, P. A. Pavlou, and R. D. Banker. Does Information and Communication Technology Lead to the Well-Being of Nations? A Country-level Empirical Investigation. *MIS Quarterly*, 40(2):417–430, 2016. ISSN 02767783, 21629730. URL https://www.jstor.org/stable/26628913. Publisher: Management Information Systems Research Center, University of Minnesota.

[83] B. Gao, Z. He, P. Sharma, Q. Kang, D. Jevdjic, J. Deng, X. Yang, Z. Yu, and P. Zuo. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 111–126, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-41-0. URL https://www.usenix.org/conference/atc24/presentation/gao-bin-cost.

[84] F. Gao, G. Tziantzioulis, and D. Wentzlaff. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 100–113, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6938-1. doi: 10.1145/3352460.3358260. URL https://doi.org/10.1145/3352460.3358260. event-place: Columbus, OH, USA.

[85] P. Gao, L. Yu, Y. Wu, and J. Li. Low Latency RNN Inference with Cellular Batching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys

’18, Porto, Portugal, 2018. Association for Computing Machinery. ISBN 978-1-4503-5584-1. doi: 10.1145/3190508.3190541. URL https://doi.org/10.1145/3190508.3190541.

[86] G. Gerganov and D. Devesa. llama.cpp, Mar. 2023. URL https://github.com/ggml-org/llama.cpp.

[87] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer. AI and Memory Wall. *IEEE Micro*, 44(3):33–39, 2024. doi: 10.1109/MM.2024.3373763.

[88] D. Giri, P. Mantovani, and L. P. Carloni. Accelerators and Coherence: An SoC Perspective. *IEEE Micro*, 38(6):36–45, 2018. doi: 10.1109/MM.2018.2877288.

[89] S. Go, H. Lee, J. Kim, J. Lee, M. K. Yoon, and W. W. Ro. Early-Adaptor: An Adaptive Framework for Proactive UVM Memory Management. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 248–258, 2023. doi: 10.1109/ISPASS57527.2023.00032.

[90] A. Gonzalez, A. Kolli, S. Khan, S. Liu, V. Dadu, S. Karandikar, J. Chang, K. Asanovic, and P. Ranganathan. Profiling Hyperscale Big Data Processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA ’23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589082. URL https://doi.org/10.1145/3579371.3589082. event-place: Orlando, FL, USA.

[91] D. Gouk, S. Kang, H. Bae, E. Ryu, S. Lee, D. Kim, J. Jang, and M. Jung. Breaking Barriers: Expanding GPU Memory with Sub-Two Digit Nanosecond Latency CXL Controller. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage ’24, pages 108–115, New York, NY, USA, 2024. Association for Computing Machinery. ISBN

9798400706301. doi: 10.1145/3655038.3665953. URL https://doi.org/10.1145/3655038.3665953. event-place: Santa Clara, CA, USA.

[92] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro*, 32(5):38–51, 2012. doi: 10.1109/MM.2012.51.

[93] Y. Gu, A. Khadem, S. Umesh, N. Liang, X. Servot, O. Mutlu, R. Iyer, and R. Das. PIM Is All You Need: A CXL-Enabled GPU-Free System for Large Language Model Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, pages 862–881, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400710797. doi: 10.1145/3676641.3716267. URL https://doi.org/10.1145/3676641.3716267. event-place: Rotterdam, Netherlands.

[94] C. Guo, J. Tang, W. Hu, J. Leng, C. Zhang, F. Yang, Y. Liu, M. Guo, and Y. Zhu. OliVe: Accelerating Large Language Models via Hardware-Friendly Outlier-Victim Pair Quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589038. URL https://doi.org/10.1145/3579371.3589038. event-place: Orlando, FL, USA.

[95] S. Gupta and S. Dwarkadas. RELIEF: Relieving Memory Pressure In SoCs Via Data Movement-Aware Accelerator Scheduling. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1063–1079, 2024. doi: 10.1109/HPCA57654.2024.00084.

[96] S. Gupta and S. Dwarkadas. Improving the Performance of Out-of-Core LLM

Inference Using Heterogeneous Host Memory. In *2025 IEEE International Symposium on Workload Characterization (IISWC)*, Irvine, CA, USA, 2025.

[97] S. Gupta, N. Madan, S. Puthoor, N. Jayasena, and S. Dwarkadas. Concurrent PIM and Load/Store Servicing in PIM-Enabled Memory. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 320–334, 2025. doi: 10.1109/ISPASS64960.2025.00037.

[98] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In J. Nieh and C. A. Waldspurger, editors, *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011*. USENIX Association, 2011. URL https://www.usenix.org/conference/usenixatc11/pegasus-coordinated-scheduling-virtualized-accelerator-based-systems.

[99] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access*, 10:52565–52608, 2022. doi: 10.1109/ACCESS.2022.3174101.

[100] R. Hadidi, B. Asgari, B. A. Mudassar, S. Mukhopadhyay, S. Yalamanchili, and H. Kim. Demystifying the characteristics of 3D-stacked memories: A case study for Hybrid Memory Cube. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 66–75, 2017. doi: 10.1109/IISWC.2017.8167757.

[101] H. Ham, J. Hong, G. Park, Y. Shin, O. Woo, W. Yang, J. Bae, E. Park, H. Sung, E. Lim, and G. Kim. Low-Overhead General-Purpose Near-Data Processing in CXL Memory Expanders. In *2024 57th IEEE/ACM International Symposium on*

Microarchitecture (MICRO), pages 594–611, 2024. doi: 10.1109/MICRO61859.2024.00051.

[102] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1815968. URL https://doi.org/10.1145/1815961.1815968. event-place: Saint-Malo, France.

[103] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pages 1135–1143, Cambridge, MA, USA, 2015. MIT Press. event-place: Montreal, Canada.

[104] R. Hankins, G. Chinya, J. Collins, P. Wang, R. Rakvic, H. Wang, and J. Shen. Multiple Instruction Stream Processor. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 114–127, 2006. doi: 10.1109/ISCA.2006.29.

[105] C. Harris and M. Stephens. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151, 1988.

[106] M. Harris. GPU Pro Tip: CUDA 7 Streams Simplify Concurrency, Jan. 2015. URL https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

[107] M. Harris. Unified Memory for CUDA Beginners, June 2017. URL https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[108] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. N. Vijaykumar. Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 372–385, 2020. doi: 10.1109/MICRO50266.2020.00040.

[109] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics*, 33(4): 1–11, July 2014. ISSN 0730-0301, 1557-7368. doi: 10.1145/2601097.2601174. URL https://dl.acm.org/doi/10.1145/2601097.2601174.

[110] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 0-12-383872-X.

[111] G. Heo, S. Lee, J. Cho, H. Choi, S. Lee, H. Ham, G. Kim, D. Mahajan, and J. Park. NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pages 722–737, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703867. doi: 10.1145/3620666.3651380. URL https://doi.org/10.1145/3620666.3651380. event-place: La Jolla, CA, USA.

[112] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems Using Integer Linear Programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 875–890. Association for Computing Machinery, New York,

NY, USA, 2020. ISBN 978-1-4503-7102-5. URL https://doi.org/10.1145/3373376.3378465.

[113] A. Ho, T. Besiroglu, E. Erdil, D. Owen, R. Rahman, Z. C. Guo, D. Atkinson, N. Thompson, and J. Sevilla. Algorithmic progress in language models, 2024. URL https://arxiv.org/abs/2403.05812. arXiv: 2403.05812.

[114] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, Nov. 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

[115] J. Hong, S. Cho, G. Park, W. Yang, Y.-H. Gong, and G. Kim. Bandwidth-Effective DRAM Cache for GPU s with Storage-Class Memory. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 139–155, 2024. doi: 10.1109/HPCA57654.2024.00021.

[116] K. S. Hong and J. Y. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41(10):1326–1331, 1992. doi: 10.1109/12.166609.

[117] C. Hooper, S. Kim, H. Mohammadzadeh, M. W. Mahoney, Y. S. Shao, K. Keutzer, and A. Gholami. KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization, 2024. URL https://arxiv.org/abs/2401.18079. arXiv: 2401.18079.

[118] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: training neural networks with low precision weights and activations. *J. Mach. Learn. Res.*, 18(1):6869–6898, Jan. 2017. ISSN 1532-4435. Publisher: JMLR.org.

[119] M. A. Ibrahim and S. Aga. Pimacolaba: Collaborative Acceleration for FFT on Commercial Processing-In-Memory Architectures. In *Proceedings of the 2024 International Symposium on Memory Systems*, MEMSYS '24, New York, NY,

USA, 2024. Association for Computing Machinery. event-place: Washington DC, DC, USA.

[120] M. Imani, Y. Kim, and T. Rosing. MPIM: Multi-purpose in-memory processing using configurable resistive memory. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 757–763, 2017. doi: 10.1109/ASPDAC.2017.7858415.

[121] Intel. Intel Optane DC Persistent Memory Product Brief, 2019. URL https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf.

[122] Intel. DDR4/DDR-T DIMM Memory Interface, June 2020. URL https://www.intel.com/content/www/us/en/docs/programmable/683867/current/ddr4-ddr-t-dimm-memory-interface.html.

[123] Intel. Intel Data Streaming Accelerator Architecture Specification, Sept. 2022. URL https://cdrdv2-public.intel.com/671116/341204-intel-data-streaming-accelerator-spec.pdf.

[124] Intel. Intel Reports Second-Quarter 2022 Financial Results, July 2022. URL https://www.intc.com/news-events/press-releases/detail/1563/intel-reports-second-quarter-2022-financial-results.

[125] Intel. Migration from Direct-Attached Intel Optane Persistent Memory to CXL-Attached Memory, Nov. 2022. URL https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-11/optane-pmem-to-cxl-tech-brief.pdf.

[126] Intel. Accelerate Artificial Intelligence Workloads with Intel Advanced Matrix Extensions, June 2024. URL https://cdrdv2.intel.com/v1/

dl/getContent/785250?fileName=Intel+AMX+Technology+Brief.pdf.

[127] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Mar. 2025.

[128] I. R. Ivanov, O. Zinenko, J. Domke, T. Endo, and W. S. Moses. Retargeting and Respecializing GPU Workloads for Performance Portability. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 119–132, Los Alamitos, CA, USA, Mar. 2024. IEEE Computer Society. doi: 10.1109/CGO57630.2024.10444828. URL https://doi.ieeecomputersociety.org/10.1109/CGO57630.2024.10444828.

[129] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv:1903.05714 [cs]*, Aug. 2019. URL http://arxiv.org/abs/1903.05714. arXiv: 1903.05714.

[130] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *2012 Symposium on VLSI Technology (VLSIT)*, pages 87–88, 2012. doi: 10.1109/VLSIT.2012.6242474.

[131] JEDEC. High Bandwidth Memory (HBM) DRAM (JESD235D), Mar. 2021.

[132] JEDEC. Graphics Double Data Rate (GDDR6) SGRAM Standard (JESD250D), May 2023.

[133] D. Jeong, J. Park, and J. Kim. Demand MemCpy: Overlapping of Computation and Data Transfer for Heterogeneous Computing. *IEEE Access*, 10:79925–79938, 2022. doi: 10.1109/ACCESS.2022.3195271.

[134] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei. S^3: Increasing GPU Utilization during Generative Inference for Higher Throughput. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 18015–18027. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/3a13be0c5dae69e0f08065f113fb10b8-Paper-Conference.pdf.

[135] Z. Jin. The Rodinia Benchmark Suite in SYCL. June 2020. doi: 10.2172/1631460. URL https://www.osti.gov/biblio/1631460.

[136] Z. Jin and J. S. Vetter. Evaluating Unified Memory Performance in HIP. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 562–568, 2022. doi: 10.1109/IPDPSW55747.2022.00096.

[137] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das. Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 1–8, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 978-1-4503-2766-4. doi: 10.1145/2588768.2576780. URL https://doi.org/10.1145/2588768.2576780. event-place: Salt Lake City, UT, USA.

[138] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das. Anatomy of GPU Memory System for Multi-Application Execution. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS ’15, pages 223–234, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-3604-8. doi: 10.1145/2818950.2818979. URL https://doi.org/10.1145/2818950.2818979. event-place: Washington DC, DC, USA.

[139] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim. HBM (High Bandwidth Memory) DRAM Technology and Architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, 2017. doi: 10.1109/IMW.2017.7939084.

[140] M. Jung. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, pages 45–51, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9399-7. doi: 10.1145/3538643.3539745. URL https://doi.org/10.1145/3538643.3539745. event-place: Virtual Event.

[141] M. Kagan, S. Gochman, D. Orenstein, and D. Lin. MMX Microarchitecture of Pentium Processors With MMX Technology and Pentium II Microprocessors. *Intel Technology Journal*, 1997.

[142] A. K. Kamath, R. Prabhu, J. Mohan, S. Peter, R. Ramjee, and A. Panwar. POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, pages 897–912, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400710797. doi: 10.1145/3676641.3715996. URL https://doi.org/10.1145/3676641.3715996. event-place: Rotterdam, Netherlands.

[143] S. Karim, J. Wünsche, M. Kuhn, G. Saake, and D. Broneske. NVM in Data Storage: A Post-Optane Future. *ACM Trans. Storage*, Apr. 2025. ISSN 1553-3077. doi: 10.1145/3731454. URL https://doi.org/10.1145/3731454. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[144] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers. Accel-Sim: An Extensible

Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020. doi: 10.1109/ISCA45697.2020.00047.

[145] C. H. Kim, W. J. Lee, Y. Paik, K. Kwon, S. Y. Kim, I. Park, and S. W. Kim. Silent-PIM: Realizing the Processing-in-Memory Computing With Standard Memory Requests. *IEEE Transactions on Parallel and Distributed Systems*, 33(2):251–262, 2022. doi: 10.1109/TPDS.2021.3065365.

[146] J. S. Kim, D. Senol Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu. GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC Genomics*, 19 (2):89, May 2018. ISSN 1471-2164. doi: 10.1186/s12864-018-4460-0. URL https://doi.org/10.1186/s12864-018-4460-0.

[147] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010. doi: 10.1109/HPCA.2010.5416658.

[148] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76, 2010. doi: 10.1109/MICRO.2010.51.

[149] Y.-B. Kim and T. Chen. Assessing merged DRAM/logic technology. In *1996 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 133–136 vol.4, 1996. doi: 10.1109/ISCAS.1996.541917.

[150] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel. Hardware Acceleration in the IBM PowerEN Processor: Architecture and Performance. In *Proceedings of the 21st International Conference on Parallel Archi-*

tectures and Compilation Techniques, PACT '12, pages 389–400, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370872. URL https://doi.org/10.1145/2370816.2370872. event-place: Minneapolis, Minnesota, USA.

[151] S. Kumar. Fundamental Limits to Moore's Law, 2015. URL https://arxiv.org/abs/1511.05956. arXiv: 1511.05956.

[152] S. Kumar, A. Shriraman, and N. Vedula. Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 733–745, Portland, Oregon, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-3402-0. doi: 10.1145/2749469.2750421. URL https://doi.org/10.1145/2749469.2750421.

[153] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. MAGIC—Memristor-Aided Logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(11):895–899, 2014. doi: 10.1109/TCSII.2014.2357292.

[154] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165. event-place: Koblenz, Germany.

[155] Y. Kwon, K. Vladimir, N. Kim, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An, J. Kim, J. Lee, I. Kim, J. Park, C. Park, Y. Song, B. Yang, H. Lee, S. Kim, D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka,

K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, M. Lee, M. Shin, M. Shin, J. Cha, C. Jung, K. Chang, C. Jeong, E. Lim, I. Park, J. Chun, and S. Hynix. System Architecture and Software Stack for GDDR6-AiM. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–25, 2022. doi: 10.1109/HCS55958.2022.9895629.

[156] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H.-S. Shin, J. Kim, B. Phuah, H. Kim, M. J. Song, A. Choi, D. Kim, S. Kim, E.-B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, and N. S. Kim. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 64, pages 350–352, 2021. doi: 10.1109/ISSCC42613.2021.9365862.

[157] G. Kyriazis. Heterogeneous System Architecture: A Technical Review. Technical report, AMD, Aug. 2012.

[158] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, 2013. doi: 10.1109/ISPASS.2013.6557176.

[159] Y. LeCun, J. Denker, and S. Solla. Optimal Brain Damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989. URL https://proceedings.neurips.cc/paper_files/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf.

[160] D. Lee, B. Hyun, T. Kim, and M. Rhu. PIM-MMU: A Memory Management Unit for Accelerating Data Transfers in Commercial PIM Systems. In

2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 627–642, 2024. doi: 10.1109/MICRO61859.2024.00053.

[161] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56, 2021. doi: 10.1109/ISCA52012.2021.00013.

[162] S. Lee, S. Lee, M. Seo, C. Park, H.-J. Lee, W. Shin, and H. Kim. A High-Performance Scheduling Algorithm for Mode Transition in PIM. In *2021 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pages 1–4, 2021. doi: 10.1109/ICCE-Asia53811.2021.9641988.

[163] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, K. Vladimir, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, J. Lee, D. Ko, Y. Jun, K. Cho, I. Kim, C. Song, C. Jeong, D. Kwon, J. Jang, I. Park, J. Chun, and J. Cho. A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 65, pages 1–3, 2022. doi: 10.1109/ISSCC42614.2022.9731711.

[164] W. J. Lee, C. H. Kim, Y. Paik, J. Park, I. Park, and S. W. Kim. Design of Processing-“Inside”-Memory Optimized for DRAM Behaviors. *IEEE Access*, 7:82633–82648, 2019. doi: 10.1109/ACCESS.2019.2924240.

[165] O. Leitersdorf, R. Ronen, and S. Kvatinsky. PyPIM: Integrating Digital Processing-in-Memory from Microarchitectural Design to Python Tensors. In

2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1632–1647, 2024. doi: 10.1109/MICRO61859.2024.00119.

[166] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 49–63, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304044. URL https://doi.org/10.1145/3297858.3304044. event-place: Providence, RI, USA.

[167] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 574–587, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 978-1-4503-9916-6. doi: 10.1145/3575693.3578835. URL https://doi.org/10.1145/3575693.3578835. event-place: Vancouver, BC, Canada.

[168] J. Li and J. Du. Study on panoramic image stitching algorithm. In *2010 Second Pacific-Asia Conference on Circuits, Communications and System*, volume 1, pages 417–420, 2010. doi: 10.1109/PACCS.2010.5626602.

[169] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 267–278, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 978-1-60558-526-0. doi: 10.1145/1555754.1555789.

URL https://doi.org/10.1145/1555754.1555789. event-place: Austin, TX, USA.

[170] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, Williamsburg VA USA, Mar. 2018. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173191. URL https://dl.acm.org/doi/10.1145/3173162.3173191.

[171] Z. Lin, M. Alshboul, Y. Solihin, and H. Zhou. Exploring Memory Persistency Models for GPUs. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 311–323, 2019. doi: 10.1109/PACT.2019.00032.

[172] Linux Kernel Organization. PCI Peer-to-Peer DMA Support. URL https://www.kernel.org/doc/html/latest/driver-api/pci/p2pdma.html.

[173] Q. Liu. Smooth Stitching Method of VR Panoramic Image Based on Improved SIFT Algorithm. In *Proceedings of the Asia Conference on Electrical, Power and Computer Engineering*, EPCE '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9612-7. doi: 10.1145/3529299.3531499. URL https://doi.org/10.1145/3529299.3531499. event-place: Shanghai, China.

[174] Y. Liu, T. Rogers, and C. Hughes. Unified Memory: GPGPU-Sim/UVM Smart Integration. Technical report, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), Feb. 2022. URL https://www.osti.gov/biblio/1844477.

[175] Z. Liu, B. Oguz, C. Zhao, E. Chang, P. Stock, Y. Mehdad, Y. Shi, R. Krishnamoorthi, and V. Chandra. LLM-QAT: Data-Free Quantization Aware Training for Large Language Models, 2023. URL https://arxiv.org/abs/2305.17888. arXiv: 2305.17888.

[176] Z. Liu, J. Yuan, H. Jin, S. H. Zhong, Z. Xu, V. Braverman, B. Chen, and X. Hu. KIVI: a tuning-free asymmetric 2bit quantization for KV cache. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024. Place: Vienna, Austria.

[177] LLVM Project. LLVM 3.8.1, July 2016. URL https://releases.llvm.org/3.8.1/docs/index.html.

[178] L. B. Lucy. An iterative technique for the rectification of observed distributions. *Astronomical Journal*, 79:745, June 1974. doi: 10.1086/111605.

[179] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 978-1-60558-798-1. doi: 10.1145/1669112.1669121. URL https://doi.org/10.1145/1669112.1669121.

[180] X. Ma, G. Fang, and X. Wang. LLM-pruner: on the structural pruning of large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc. event-place: New Orleans, LA, USA.

[181] S. Madabusi and S. V. Gangashetty. Edge detection for facial images under noisy conditions. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, pages 2689–2693, 2012.

[182] S. Malardel, N. Wedi, W. Deconinck, M. Diamantakis, C. Kühnlein, G. Mozdzynski, M. Hamrud, and P. Smolarkiewicz. A new grid for the IFS. *ECMWF Newsletter*, 146:23–28, Jan. 2016.

[183] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E. G. Cota, M. Petracca, C. Pilato, and L. P. Carloni. Agile SoC Development with Open ESP. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-8026-3. doi: 10.1145/3400302.3415753. URL https://doi.org/10.1145/3400302.3415753. event-place: Virtual Event, USA.

[184] M. Marazzi, T. Sachsenweger, F. Solt, P. Zeng, K. Takashi, M. Yarema, and K. Razavi. HiFi-DRAM: Enabling High-fidelity DRAM Research by Uncovering Sense Amplifiers with IC Imaging. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 133–149, 2024. doi: 10.1109/ISCA59077.2024.00020.

[185] P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka. DRAGON: Breaking GPU Memory Capacity Limits with Direct NVM Access. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 414–426, 2018. doi: 10.1109/SC.2018.00035.

[186] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, pages 742–755, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 978-1-4503-9918-0. doi: 10.1145/3582016.3582063. URL https://doi.org/10.1145/3582016.3582063. event-place: Vancouver, BC, Canada.

[187] N. McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 7(2):188–201, 1999. doi: 10.1109/90.769767.

[188] K. Menychtas, K. Shen, and M. L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 301–316, Salt Lake City, Utah, USA, 2014. Association for Computing Machinery. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541963. URL https://doi.org/10.1145/2541940.2541963.

[189] Meta. Introducing Llama 3.1: Our most capable models to date, July 2024. URL https://ai.meta.com/blog/meta-llama-3-1/.

[190] Meta. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation, Apr. 2025. URL https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

[191] G. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. doi: 10.1109/JPROC.1998.658762.

[192] O. Mutlu. Main Memory Scaling: Challenges and Solution Directions. In R. O. Topaloglu, editor, *More than Moore Technologies for Next Generation Computer Design*, pages 127–153. Springer New York, New York, NY, 2015. ISBN 978-1-4939-2163-8. doi: 10.1007/978-1-4939-2163-8_6. URL https://doi.org/10.1007/978-1-4939-2163-8_6.

[193] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 146–160, 2007. doi: 10.1109/MICRO.2007.21.

[194] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *2008 International Symposium on Computer Architecture*, pages 63–74, 2008. doi: 10.1109/ISCA.2008.7.

[195] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun. A Modern Primer on Processing in Memory, 2022. arXiv: 2012.03112.

[196] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das. VIP: Virtualizing IP Chains on Handheld Platforms. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 655–667, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-3402-0. doi: 10.1145/2749469.2750382. URL https://doi.org/10.1145/2749469.2750382. event-place: Portland, Oregon.

[197] A. Nag and R. Balasubramonian. OrderLight: Lightweight Memory-Ordering Primitive for Efficient Fine-Grained PIM Computations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pages 298–310, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8557-2. doi: 10.1145/3466752.3480103. URL https://doi.org/10.1145/3466752.3480103. event-place: Virtual Event, Greece.

[198] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 457–468, 2017. doi: 10.1109/HPCA.2017.54.

[199] NVIDIA. NVIDIA NVLink and NVLink Switch. URL https://www.nvidia.com/en-us/data-center/nvlink/.

[200] NVIDIA. NVIDIA Deep Learning Accelerator (NVDLA), 2017. URL www.nvdla.org.

[201] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture, 2020.

[202] NVIDIA. nvbandwidth, Apr. 2022. URL https://github.com/NVIDIA/nvbandwidth.

[203] NVIDIA. CUDA Runtime API, June 2023. URL https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf.

[204] NVIDIA. NVIDIA GH200 Grace Hopper Superchip Architecture, 2023. URL https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper.

[205] NVIDIA. CUDA Toolkit Documentation: Stream Management, Apr. 2024. URL https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html.

[206] NVIDIA. Multi-Instance GPU, Aug. 2024. URL https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[207] NVIDIA. Multi-Process Service, June 2024. URL https://docs.nvidia.com/deploy/mps/index.html.

[208] S. D. Oliner and D. E. Sichel. Information technology and productivity: where are we now and where are we going? *Journal of Policy Modeling*, 25(5):477–503, 2003. ISSN 0161-8938. doi: https://doi.org/10.1016/S0161-8938(03)00042-5. URL https://www.sciencedirect.com/science/article/pii/S0161893803000425.

[209] S. Pandey, A. K. Kamath, and A. Basu. GPM: Leveraging Persistent Memory from a GPU. In *Proceedings of the Twenty-Seventh International Confer-*

ence on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’22, Lausanne, Switzerland, 2022. Association for Computing Machinery. ISBN 978-1-4503-9205-1. doi: 10.1145/3503222.3507758. URL https://doi.org/10.1145/3503222.3507758.

[210] S. Pandey, A. K. Kamath, and A. Basu. Scoped Buffered Persistency Model for GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 688–701, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 978-1-4503-9916-6. doi: 10.1145/3575693.3575749. URL https://doi.org/10.1145/3575693.3575749. event-place: Vancouver, BC, Canada.

[211] J. Park, D. Jeong, and J. Kim. UVMMU: Hardware-Offloaded Page Migration for Heterogeneous Computing. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023. doi: 10.23919/DATE56975.2023.10137307.

[212] J. Park, J. Choi, K. Kyung, M. J. Kim, Y. Kwon, N. S. Kim, and J. H. Ahn. AttAcc! Unleashing the Power of PIM for Batched Transformer-based Generative Model Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS ’24, pages 103–119, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640422. URL https://doi.org/10.1145/3620665.3640422. event-place: La Jolla, CA, USA.

[213] P. Patel, E. Choukse, C. Zhang, A. Shah, I. Goiri, S. Maleki, and R. Bianchini. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132, 2024. doi: 10.1109/ISCA59077.2024.00019.

[214] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang. Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using intel optane DC persistent memory modules. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, pages 288–303, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-7206-0. doi: 10.1145/3357526.3357541. URL https://doi.org/10.1145/3357526.3357541. event-place: Washington, District of Columbia, USA.

[215] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. Scheduling techniques for GPU architectures with processing-in-memory capabilities. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 31–44, 2016. doi: 10.1145/2967938.2967940.

[216] S. Pei and R. Pitchumani. CMM-H (CXL Memory Module – Hybrid): Rethinking Storage for the Memory-Centric Computing Era, Dec. 2023. URL https://semiconductor.samsung.com/us/news-events/tech-blog/rethinking-storage-for-the-memory-centric-computing-era/.

[217] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 137–151, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304025. URL https://doi.org/10.1145/3297858.3304025. event-place: Providence, RI, USA.

[218] I. B. Peng, M. B. Gokhale, and E. W. Green. System evaluation of the In-

tel optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems - MEMSYS '19*, pages 304–315, Washington, District of Columbia, 2019. ACM Press. ISBN 978-1-4503-7206-0. doi: 10.1145/3357526.3357568. URL http://dl.acm.org/citation.cfm?doid=3357526.3357568.

[219] S. Puthoor, X. Tang, J. Gross, and B. M. Beckmann. Oversubscribed Command Queues in GPUs. In *Proceedings of the 11th Workshop on General Purpose GPUs*, GPGPU-11, pages 50–60, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 978-1-4503-5647-3. doi: 10.1145/3180270.3180271. URL https://doi.org/10.1145/3180270.3180271. event-place: Vienna, Austria.

[220] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. Horowitz. Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing. *Commun. ACM*, 58(4):85–93, Mar. 2015. ISSN 0001-0782. doi: 10.1145/2735841. URL https://doi.org/10.1145/2735841. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[221] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 978-1-60558-526-0. doi: 10.1145/1555754.1555760. URL https://doi.org/10.1145/1555754.1555760. event-place: Austin, TX, USA.

[222] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

[223] D. Ralph and M. Stiles. Spin transfer torques. *Journal of Magnetism and Magnetic Materials*, 320(7):1190–1216, Apr. 2008. ISSN 0304-8853. doi: 10.1016/j.jmmm.2007.12.019. URL http://dx.doi.org/10.1016/j.jmmm.2007.12.019. Publisher: Elsevier BV.

[224] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 978-1-4503-0102-2. doi: 10.1145/1995896.1995911. URL https://doi.org/10.1145/1995896.1995911. event-place: Tucson, Arizona, USA.

[225] B. R. Rau. Pseudo-randomly interleaved memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 74–83, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0-89791-394-9. doi: 10.1145/115952.115961. URL https://doi.org/10.1145/115952.115961. event-place: Toronto, Ontario, Canada.

[226] M. Ravanelli, P. Brakel, M. Omologo, and Y. Bengio. Light Gated Recurrent Units for Speech Recognition. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(2):92–102, 2018. doi: 10.1109/TETCI.2017.2762739.

[227] V. J. Reddi. Mobile SoCs: The Wild West of Domain Specific Architectures, Sept. 2018. URL https://www.sigarch.org/mobile-socs/.

[228] V. J. Reddi and M. D. Hill. Accelerator-Level Parallelism (ALP), Sept. 2019. URL https://www.sigarch.org/accelerator-level-parallelism/.

[229] V. J. Reddi, H. Yoon, and A. Knies. Two Billion Devices and Counting. *IEEE Micro*, 38(1):6–21, 2018. doi: 10.1109/MM.2018.011441560.

[230] J. Ren, J. Luo, I. Peng, K. Wu, and D. Li. Optimizing Large-Scale Plasma Simulations on Persistent Memory-Based Heterogeneous Memory with Effective Data Placement across Memory Hierarchy. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, pages 203–214, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8335-6. doi: 10.1145/3447818.3460356. URL https://doi.org/10.1145/3447818.3460356. event-place: Virtual Event, USA.

[231] S. Rennich. CUDA C/C++ Streams and Concurrency. URL https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[232] S. Resch, S. K. Khatamifard, Z. I. Chowdhury, M. Zabihi, Z. Zhao, H. Cilasun, J.-P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu. MOUSE: Inference In Non-volatile Memory for Energy Harvesting Applications. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 400–414, 2020. doi: 10.1109/MICRO50266.2020.00042.

[233] S. Resch, H. Cilasun, Z. Chowdhury, M. Zabihi, Z. Zhao, J.-P. Wang, S. Sapatnekar, and U. R. Karpuzcu. On Endurance of Processing in (Nonvolatile) Memory. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589114. URL https://doi.org/10.1145/3579371.3589114. event-place: Orlando, FL, USA.

[234] W. H. Richardson. Bayesian-Based Iterative Method of Image Restoration. *J. Opt. Soc. Am.*, 62(1):55–59, Jan. 1972. doi: 10.1364/JOSA.62.000055. URL http://www.osapublishing.org/abstract.cfm?URI=josa-62-1-55. Publisher: OSA.

[235] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1-58113-232-8. doi: 10.1145/339647.339668. URL https://doi.org/10.1145/339647.339668. event-place: Vancouver, British Columbia, Canada.

[236] S. Rogers, J. Slycord, M. Baharani, and H. Tabkhi. gem5-SALAM: A System Architecture for LLVM-based Accelerator Modeling. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 471–482, 2020. doi: 10.1109/MICRO50266.2020.00047.

[237] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, Cascais, Portugal, 2011. Association for Computing Machinery. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043579. URL https://doi.org/10.1145/2043556.2043579.

[238] K. I. Roumeliotis and N. D. Tselikas. ChatGPT and Open-AI Models: A Preliminary Review. *Future Internet*, 15(6), 2023. ISSN 1999-5903. doi: 10.3390/fi15060192. URL https://www.mdpi.com/1999-5903/15/6/192.

[239] J. Ru, Y. Yang, J. Grundy, J. Keung, and L. Hao. An efficient deadline constrained and data locality aware dynamic scheduling framework for multitenancy clouds. *Concurrency and Computation: Practice and Experience*, 33(5):e6037, 2021. doi: https://doi.org/10.1002/cpe.6037. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6037.

[240] M. Rybczynska. Device-to-device memory-transfer offload with P2PDMA, Oct. 2018. URL https://lwn.net/Articles/767281/.

[241] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill. Parallel Real-Time Scheduling of DAGs. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014. doi: 10.1109/TPDS.2013.2297919.

[242] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov. System-on-Chip: Reuse and Integration. *Proceedings of the IEEE*, 94(6):1050–1069, 2006. doi: 10.1109/JPROC.2006.873611.

[243] Samsung. Samsung CXL Solutions – CMM-H, Mar. 2024. URL https://semiconductor.samsung.com/us/news-events/tech-blog/samsung-cxl-solutions-cmm-h/.

[244] G. Sanderson. 3Blue1Brown: Neural Networks, 2024. URL https://www.3blue1brown.com/topics/neural-networks.

[245] J. Sankaran and N. Zoran. TDA2X, a SoC optimized for advanced driver assistance systems. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2204–2208, 2014. doi: 10.1109/ICASSP.2014.6853990.

[246] M. C. d. Santos, T. Jia, M. Cochet, K. Swaminathan, J. Zuckerman, P. Mantovani, D. Giri, J. J. Zhang, E. J. Loscalzo, G. Tombesi, K. Tien, N. Chandramoorthy, J.-D. Wellman, D. Brooks, G.-Y. Wei, K. Shepard, L. P. Carloni, and P. Bose. A Scalable Methodology for Agile Chip Development with Open-Source Hardware Components. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9217-4. doi: 10.1145/3508352.3561102. URL https://doi.org/10.1145/3508352.3561102. event-place: San Diego, California.

[247] T. C. Schroeder. Peer-to-Peer & Unified Virtual Addressing, 2011. URL https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf.

[248] M. Seo, X. T. Nguyen, S. J. Hwang, Y. Kwon, G. Kim, C. Park, I. Kim, J. Park, J. Kim, W. Shin, J. Won, H. Choi, K. Kim, D. Kwon, C. Jeong, S. Lee, Y. Choi, W. Byun, S. Baek, H.-J. Lee, and J. Kim. IANUS: Integrated Accelerator based on NPU-PIM Unified Memory System. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pages 545–560, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703867. doi: 10.1145/3620666.3651324. URL https://doi.org/10.1145/3620666.3651324. event-place: La Jolla, CA, USA.

[249] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 185–197, 2013.

[250] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 273–287, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4952-9. doi: 10.1145/3123939.3124544. URL https://doi.org/10.1145/3123939.3124544. event-place: Cambridge, Massachusetts.

[251] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks. Co-designing

accelerators and SoC interfaces using gem5-Aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Taipei, Taiwan, Oct. 2016. IEEE. ISBN 978-1-5090-3508-3. doi: 10.1109/MICRO.2016.7783751. URL http://ieeexplore.ieee.org/document/7783751/.

[252] D. D. Sharma. Compute Express Link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 5–12, 2022. doi: 10.1109/HOTI55740.2022.00017.

[253] S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer. Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34 (05):8815–8821, Apr. 2020. ISSN 2374-3468. doi: 10.1609/aaai.v34i05.6409. URL https://ojs.aaai.org/index.php/AAAI/article/view/6409.

[254] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Y. Fu, Z. Xie, B. Chen, C. Barrett, J. E. Gonzalez, P. Liang, C. Ré, I. Stoica, and C. Zhang. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU, 2023. arXiv: 2303.06865.

[255] M. Shihab, J. Zhang, S. Gao, J. Sloan, and M. Jung. Couture: Tailoring STT-MRAM for Persistent Main Memory. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*, Savannah, GA, Nov. 2016. USENIX Association. URL https://www.usenix.org/conference/inflow16/workshop-program/presentation/shihab.

[256] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo. McDRAM: Low Latency

and Energy-Efficient Matrix Computations in DRAM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2613–2622, 2018. doi: 10.1109/TCAD.2018.2857044.

[257] R. Sikarwar, A. Agrawal, and R. S. Kushwah. An Edge Based Efficient Method of Face Detection and Feature Extraction. In *2015 Fifth International Conference on Communication Systems and Network Technologies*, pages 1147–1151, 2015. doi: 10.1109/CSNT.2015.167.

[258] Y. Song, Z. Mi, H. Xie, and H. Chen. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, pages 590–606, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi: 10.1145/3694715.3695964. URL https://doi.org/10.1145/3694715.3695964. event-place: Austin, TX, USA.

[259] H. S. Stone. A Logic-in-Memory Computer. *IEEE Transactions on Computers*, C-19(1):73–78, 1970. doi: 10.1109/TC.1970.5008902.

[260] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, May 2008. ISSN 1476-4687. doi: 10.1038/nature06932. URL https://doi.org/10.1038/nature06932.

[261] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu. BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):3071–3087, 2016. doi: 10.1109/TPDS.2016.2526003.

[262] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the*

56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '23, pages 105–121, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3614256. URL https://doi.org/10.1145/3613424.3614256. event-place: Toronto, ON, Canada.

[263] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press. event-place: Montreal, Canada.

[264] C. Tan, M. Karunaratne, T. Mitra, and L.-S. Peh. Stitch: Fusible Heterogeneous Accelerators Enmeshed with Many-Core Architecture for Wearables. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 575–587, 2018. doi: 10.1109/ISCA.2018.00054.

[265] A. Thompson and C. Newburn. GPUDirect Storage: A Direct Path Between Storage and GPU Memory, Aug. 2019. URL https://developer.nvidia.com/blog/gpudirect-storage/.

[266] N. C. Thompson and S. Spanuth. The decline of computers as a general purpose technology. *Commun. ACM*, 64(3):64–72, Feb. 2021. ISSN 0001-0782. doi: 10.1145/3430936. URL https://doi.org/10.1145/3430936. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[267] N. C. Thompson, S. Ge, and G. F. Manso. The Importance of (Exponentially More) Computing Power, 2022. URL https://arxiv.org/abs/2206.14007. arXiv: 2206.14007.

[268] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon. Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21,

pages 754–766, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8557-2. doi: 10.1145/3466752.3480128. URL https://doi.org/10.1145/3466752.3480128. event-place: Virtual Event, Greece.

[269] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975. ISSN 0022-0000. doi: https://doi.org/10.1016/S0022-0000(75)80008-0. URL https://www.sciencedirect.com/science/article/pii/S0022000075800080.

[270] UPMEM. UPMEM PIM Technical paper, Aug. 2022. URL https://www.upmem.com/technology/.

[271] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. *ACM Trans. Archit. Code Optim.*, 12(4), Jan. 2016. ISSN 1544-3566. doi: 10.1145/2847255. URL https://doi.org/10.1145/2847255. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[272] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is All you Need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[273] R. S. Venkatesh, T. Mason, P. Fernando, G. Eisenhauer, and A. Gavrilovska. Scheduling HPC Workflows with Intel Optane Persistent Memory. In *2021*

IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 56–65, 2021. doi: 10.1109/IPDPSW52791.2021.00017.

[274] V. Viswanathan, K. Kumar, T. Willhalm, S. Sakthivelu, and S. Srikanthan. Intel Memory Latency Checker, May 2024. URL https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[275] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter. Exploring hybrid memory for GPU energy efficiency through software-hardware co-design. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 93–102, 2013. doi: 10.1109/PACT.2013.6618807.

[276] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. BigDataBench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499, 2014. doi: 10.1109/HPCA.2014.6835958.

[277] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-Core Multithreaded System. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 156–166, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250753. URL https://doi.org/10.1145/1250734.1250753. event-place: San Diego, California, USA.

[278] S.-T. Wang, H. Xu, A. Mamandipoor, R. Mahapatra, B. H. Ahn, S. Ghodrati, K. Kailas, M. Alian, and H. Esmaeilzadeh. Data Motion Acceleration: Chain-

ing Cross-Domain Multi Accelerators. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1043–1062, 2024. doi: 10.1109/HPCA57654.2024.00083.

[279] X. Wang, J. Liu, J. Wu, S. Yang, J. Ren, B. Shankar, and D. Li. Performance Characterization of CXL Memory and Its Use Cases. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2025.

[280] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao. Characterizing and Modeling Non-Volatile Memory Systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508, 2020. doi: 10.1109/MICRO50266.2020.00049.

[281] T. Wei, N. Turtayeva, M. Orenes-Vera, O. Lonkar, and J. Balkind. Cohort: Software-Oriented Acceleration for Heterogeneous SoCs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, pages 105–117, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 978-1-4503-9918-0. doi: 10.1145/3582016.3582059. URL https://doi.org/10.1145/3582016.3582059. event-place: Vancouver, BC, Canada.

[282] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons. An Early Evaluation of Intel's Optane DC Persistent Memory Module and Its Impact on High-Performance Scientific Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6229-0. doi: 10.1145/3295500.3356159. URL https://doi.org/10.1145/3295500.3356159. event-place: Denver, Colorado.

[283] F. Wen, M. Qin, P. Gratz, and N. Reddy. Software Hint-Driven Data Management for Hybrid Memory in Mobile Systems. *ACM Trans. Embed. Comput. Syst.*, 21(1), Jan. 2022. ISSN 1539-9087. doi: 10.1145/3494536. URL https://doi.org/10.1145/3494536. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[284] Wikipedia. PCI Express link performance. URL https://en.wikipedia.org/wiki/PCI_Express#Comparison_table.

[285] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang. Pangaea: A tightly-coupled IA32 heterogeneous chip multiprocessor. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 52–61, 2008.

[286] L. Wu, R. Sharifi, M. Lenjani, K. Skadron, and A. Venkat. Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel k-mer Matching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 251–264, 2021. doi: 10.1109/ISCA52012.2021.00028.

[287] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995. ISSN 0163-5964. doi: 10.1145/216585.216588. URL https://doi.org/10.1145/216585.216588. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[288] M. Xia, T. Gao, Z. Zeng, and D. Chen. Sheared LLaMA: Accelerating Language Model Pre-training via Structured Pruning, 2024. URL https://arxiv.org/abs/2310.06694. arXiv: 2310.06694.

[289] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. SmoothQuant: accurate and efficient post-training quantization for large language models. In *Pro-*

ceedings of the 40th International Conference on Machine Learning, ICML'23. JMLR.org, 2023. Place: Honolulu, Hawaii, USA.

[290] C. Xie, S. L. Song, J. Wang, W. Zhang, and X. Fu. Processing-in-Memory Enabled Graphics Processors for 3D Rendering. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 637–648, 2017. doi: 10.1109/HPCA.2017.37.

[291] J. Xu and S. Swanson. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, USA, 2016. USENIX Association. ISBN 978-1-931971-28-7. event-place: Santa Clara, CA.

[292] Z. Xue, Y. Song, Z. Mi, X. Zheng, Y. Xia, and H. Chen. PowerInfer-2: Fast Large Language Model Inference on a Smartphone, 2024. URL https://arxiv.org/abs/2406.06282. arXiv: 2406.06282.

[293] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, Feb. 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/yang.

[294] K. Yang, M. Yang, and J. H. Anderson. Reducing Response-Time Bounds for DAG-Based Task Systems on Heterogeneous Multicore Platforms. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, pages 349–358, Brest, France, 2016. Association for Computing Machinery. ISBN 978-1-4503-4787-7. doi: 10.1145/2997465.2997486. URL https://doi.org/10.1145/2997465.2997486.

[295] S.-P. Yang, M. Kim, S. Nam, J. Park, J.-y. Choi, E. H. Nam, E. Lee, S. Lee, and B. S. Kim. Overcoming the Memory Wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 601–617, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-35-9. URL https://www.usenix.org/conference/atc23/presentation/yang-shao-peng.

[296] W. Yang, X. Zhang, Q. Lei, D. Shen, P. Xiao, and Y. Huang. Lane Position Detection Based on Long Short-Term Memory (LSTM). *Sensors (Basel, Switzerland)*, 20(11), May 2020. ISSN 1424-8220. doi: 10.3390/s20113115. Place: Switzerland.

[297] Z. Yao, R. Yazdani Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He. ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 27168–27183. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/adf7fa39d65e2983d724ff7da57f00ac-Paper-Conference.pdf.

[298] P. Yedlapalli, N. C. Nachiappan, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. Short-Circuiting Memory Traffic in Handheld Platforms. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 166–177, 2014. doi: 10.1109/MICRO.2014.60.

[299] T. T. Yeh, M. Sinclair, B. M. Beckmann, and T. G. Rogers. Deadline-Aware Offloading for High-Throughput Accelerators. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[300] Y. J. Yoon, N. Concer, M. Petracca, and L. P. Carloni. Virtual Channels and Mul-

tiple Physical Networks: Two Alternatives to Improve NoC Performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32 (12):1906–1919, 2013. doi: 10.1109/TCAD.2013.2276399.

[301] Y. Yuan, R. Wang, N. Ranganathan, N. Rao, S. Kumar, P. Lantz, V. Sanjeepan, J. Cabrera, A. Kwatra, R. Sankaran, I. Jeong, and N. S. Kim. Intel Accelerators Ecosystem: An SoC-Oriented Perspective : Industry Product. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 848–862, 2024. doi: 10.1109/ISCA59077.2024.00066.

[302] A. Zadeh, I. Edo, O. Awad, and A. Moshovos. GOBO: Quantizing Attention-Based NLP Models for Low Latency and Energy Efficient Inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 811–824, Los Alamitos, CA, USA, Oct. 2020. IEEE Computer Society. doi: 10.1109/MICRO50266.2020.00071. URL https://doi.ieeecomputersociety.org/10.1109/MICRO50266.2020.00071.

[303] H.-E. Zahaf, N. Capodieci, R. Cavicchioli, M. Bertogna, and G. Lipari. *A C-DAG task model for scheduling complex real-time tasks on heterogeneous platforms: preemption matters*. arXiv, 2019. arXiv: 1901.02450.

[304] H. Zhang, P. V. Rengasamy, N. C. Nachiappan, S. Zhao, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. FLOSS: FLOw Sensitive Scheduling on Mobile Platforms. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018. doi: 10.1109/DAC.2018.8465877.

[305] H. Zhang, Y. Zhou, Y. Xue, Y. Liu, and J. Huang. G10: Enabling An Efficient Unified GPU Memory and Storage Architecture with Smart Tensor Migrations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’23, pages 395–410, New York, NY, USA,

2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3614309. URL https://doi.org/10.1145/3613424.3614309. event-place: Toronto, ON, Canada.

[306] J. Zhang, D. Donofrio, J. Shalf, M. T. Kandemir, and M. Jung. NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 13–24, 2015. doi: 10.1109/PACT.2015.43.

[307] J. Zhang, N. Beckwith, and J. J. Li. GORDON: Benchmarking Optane DC Persistent Memory Modules on FPGAs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 97–105, 2021. doi: 10.1109/FCCM51124.2021.00019.

[308] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer. OPT: Open Pre-trained Transformer Language Models, 2022. URL https://arxiv.org/abs/2205.01068. arXiv: 2205.01068.

[309] Y. Zhao, M. Gao, F. Liu, Y. Hu, Z. Wang, H. Lin, J. Li, H. Xian, H. Dong, T. Yang, N. Jing, X. Liang, and L. Jiang. UM-PIM: DRAM-based PIM with Uniform & Shared Memory Space. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 644–659, 2024. doi: 10.1109/ISCA59077.2024.00053.

[310] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 345–357, 2016. doi: 10.1109/HPCA.2016.7446077.

A Chapter 3 Artifact Appendix

A.1 Abstract

This artifact appendix describes how to run RELIEF and other accelerator scheduling policies described in Chapter 3 using gem5. The artifact includes the implementation of all the policies in gem5 and our vision and RNN benchmark suite, along with pre-built binaries for the latter. It also includes optional instructions to rebuild the benchmark binaries and hardware models.

A.2 Artifact check-list (meta-information)

- **Algorithm:** RELIEF, a least-laxity based scheduling policy.
- **Program:** gem5 (C++ and Python code).
- **Compilation:** GCC, SCons.
- **Binary:** Vision and RNN binaries, compiled using GNU Arm Embedded toolchain v8.3.
- **Run-time environment:** Any modern Linux distribution.
- **Hardware:** x86-based CPU with 10 cores and 32GB main memory.

- **Metrics:** Data forwards, data movement, accelerator occupancy, slowdown, node deadlines met, DAG deadlines met.
- **Output:** gem5 statistics and execution trace.
- **How much disk space required (approximately)?:** 17 GB.
- **How much time is needed to prepare workflow (approximately)?:** 2-3 hours.
- **How much time is needed to complete experiments (approximately)?:** 8-10 hours.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD-3
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.10237117

A.3 Description

A.3.1 How to access

The code is available on GitHub[1] and Zenodo[2].

A.3.2 Hardware dependencies

Recent x86 based CPU with at least 10 cores and 32GB main memory. The simulations take multiple hours to run, and we recommend at least 60 cores and 150 GB of main memory to run all of them in parallel.

A.3.3 Software dependencies

- Linux OS with a recent version of GCC.

[1] https://github.com/Sacusa/gem5-SALAM/tree/HPCA_2024

[2] https://doi.org/10.5281/zenodo.10237117

- Python 2 with pip installed.
- (Optional) GNU Arm Embedded toolchain v8.3
- (Optional) LLVM 3.8

A.4 Installation

The steps below detail the installation for gem5 and associated Python dependencies. There is a step for building the benchmarks that requires GNU Arm Embedded toolchain v8.3. Our distribution already includes benchmark binaries, so this step is optional.

1. Navigate to the project root directory and install Python dependencies by running:

   ```
   pip install -r requirements.txt
   ```

2. Build gem5 by following the instructions in README.md.

3. Set `M5_PATH` environment variable:

   ```
   export M5_PATH=`pwd`
   ```

4. (Optional) Build the benchmarks by navigating to `$M5_PATH/benchmarks/scheduler/sw` and running the following command. Note that this requires the installation of GNU Arm Embedded toolchain, described in README.md.

   ```
   ./create_binary_combinations_3.sh
   ```

   The binaries will be put in the directory `bin_comb_3`.

5. (Optional) Compile the accelerator descriptions into LLVM IR by navigating to `$M5_PATH/benchmarks/scheduler/hw` and running `make`. Note that this requires the installation of LLVM 3.8, described in README.md.

A.5 Experiment workflow

Navigate to the project root directory and launch high contention scenario simulations by running:

```
./run_combinations_3.sh `nproc`
```

The simulations need at least 10 cores to finish in a reasonable period. The results will be saved in the directory
`$M5_PATH/BM_ARM_OUT/comb_3`.

A.6 Evaluation and expected results

The directory `$M5_PATH/BM_ARM_OUT/scripts/comb_3` contains plotting scripts to visualize key results from Chapter 3. Once the simulations finish, the following figures can be regenerated using the respective scripts in the directory:

- Figure 3.4c: `plot_forwards.py`
- Figure 3.5c: `plot_data_movement.py`
- Figure 3.7c: `plot_accelerator_occupancy.py`
- Figure 3.9a: `plot_slowdown.py`
- Figures 3.8c, 3.9b: `plot_deadlines_met.py`

Each script can be run as:

```
python <script>
```

The scripts use matplotlib to produce the figures in PDF format, which are stored in `$M5_PATH/BM_ARM_OUT/scripts/plots`.

B Chapter 4 Artifact Appendix

B.1 Abstract

This artifact appendix describes how to reproduce key results from Chapter 4. The artifact includes a modified version of the GPGPU-Sim 4.0.1 simulator, the PIM benchmark suite, and the Rodinia benchmark suite with input data. The modified GPGPU-Sim implements the baseline and proposed interconnect architecture and memory controller scheduling policies, along with modifications to some CUDA APIs to provide more control over concurrent kernel launches. The artifact can be set up easily using the provided Dockerfile.

B.2 Artifact check-list (meta-information)

- **Algorithm:** F3FS, a memory controller scheduling policy.
- **Program:** GPGPU-Sim, Rodinia benchmark suite, PIM benchmarks.
- **Data set:** Modified Rodinia benchmark suite inputs.
- **Hardware:** Dual core x86-64 based CPU and 8GB memory.

- **Metrics:** Fairness index, system throughput, number of mode switches, conflicts per switch, drain latency per switch, LLM speedup.
- **Output:** GPGPU-Sim output statistics and plots generated using matplotlib.
- **How much disk space required (approximately)?:** 20 GB.
- **How much time is needed to prepare workflow (approximately)?:** 15 minutes.
- **How much time is needed to complete experiments (approximately)?:** 2 weeks.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Creative Commons 4.0
- **Data licenses (if publicly available)?:** Creative Commons 4.0
- **Workflow automation framework used?:** Docker.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.15164086

B.3 Description

B.3.1 How to access

The artifact is available on both GitHub and Zenodo at the following links. We recommend using the Dockerfile to set up the artifact.

- Zenodo: https://zenodo.org/records/15164086
- GitHub:
  - Dockerfile: https://gist.github.com/Sacusa/47801d133eda38317bc8fc84013ed041
  - Simulator: https://github.com/Sacusa/GPGPU-Sim-4.0.1/tree/ISPASS_2025

- Benchmarks: https://github.com/Sacusa/PIM_apps/tree/ISPASS_2025

B.3.2 Hardware dependencies

Recent x86-64 based CPU with at least 2 cores and 8GB memory. There are a total of 3258 simulations, so more cores and memory would help.

B.3.3 Software dependencies

This artifact only requires Docker to run.

B.3.4 Data sets

Input data is generated using scripts that come with the Rodinia benchmark suite.

B.4 Installation

The steps below detail how to build a Docker image and start a container for the artifact. This assumes that the user already has Docker installed.

1. Download the Dockerfile from either GitHub or Zenodo.

2. Navigate to the downloaded file from a terminal and execute the following :

   ```
   docker build -t ispass2025:latest .
   ```

3. Once the build is complete, start a container by running the following command:

   ```
   docker run -i -t --name <name> ispass2025:latest
   ```

   Replace <*name*> with a name for the container. This command will create a new container and attach to its terminal.

4. To start the container again in the future, run:

   ```
   docker start <name>
   ```

   And connect to it by running:

   ```
   docker attach <name>
   ```

B.5 Experiment workflow

Navigate to:

```
/opt/PIM_apps/STREAM
```

and launch the script:

```
run_baseline.sh 8
```

to run the baseline PIM experiments. This will simulate PIM kernels running alone. Next, navigate to:

```
/opt/PIM_apps/rodinia_3.1_pim/cuda
```

and run the script:

```
launch_ispass2025.sh
```

to run the baseline Rodinia and LLM experiments, followed by the competitive and collaborative experiments. While the results presented in the chapter require a total of 3258 simulations, the script can be modified to run a subset of experiments for faster reproduction of results.

B.6 Evaluation and expected results

The directory:

```
/opt/PIM_apps/rodinia_3.1_pim/cuda/scripts
```

contains plotting scripts to visualize key results from Chapter 4. Once the simulations finish, the following figures can be regenerated using the respective scripts in the directory:

- Figure 4.7: `plot_mem_arrival_rate.py`
- Figure 4.9a: `plot_fairness_index.py`
- Figure 4.9b: `plot_throughput.py`
- Figure 4.10a: `plot_num_switches.py`
- Figures 4.10b, 4.10c: `plot_switch_overheads.py`
- Figure 4.11: `plot_llm_speedup.py`

C Chapter 5 Artifact Appendix

C.1 Abstract

This artifact appendix describes how to reproduce key results from Chapter 5. The artifact includes a modified version of FlexGen that implements the two proposed weight allocation schemes, along with the author collected data and helper scripts to plot the figures described in the chapter.

C.2 Artifact check-list (meta-information)

- **Algorithm:** HeLM and All-CPU.
- **Program:** FlexGen.
- **Model:** OPT-30B and OPT-175B.
- **Data set:** c4/realnewslike.
- **Hardware:** Modern x86-64 based CPU with at least 100 GB DRAM+Optane memory and a CUDA-compatible GPU with at least 4 GB memory.
- **Metrics:** Time to first token (TTFT), time between tokens (TBT), throughput (tokens/second), compute/communication latency overlap

- **How much disk space required (approximately)?:** 450 GB.
- **How much time is needed to prepare workflow (approximately)?:** 2-4 hours.
- **How much time is needed to complete experiments (approximately)?:** 4 days.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache-2.0.
- **Data licenses (if publicly available)?:** OPT-175B license (model) and Apache-2.0 (data set).
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.16905746.

C.2.1 How to access

The artifact is available on both GitHub and Zenodo at the following links..

- Zenodo: https://zenodo.org/records/16905746
- GitHub: https://github.com/Sacusa/FlexLLMGen

C.2.2 Hardware dependencies

Recent x86-64 based CPU with at least 100 GB of heterogeneous memory. Our system, for instance, combines DRAM and Intel Optane. A CUDA-compatible GPU with at least 4 GB of onboard memory.

C.2.3 Software dependencies

PyTorch >= 1.12.

C.2.4 Data sets

c4/realnewslike: https://huggingface.co/datasets/allenai/c4.

C.2.5 Models

FlexGen automatically downloads most model weights. The weights for OPT-175B can be obtained at https://huggingface.co/Neko-Institute-of-Science/OPT-175B-NumPy

C.3 Installation

1. Download the input data set and models from the links above using HuggingFace CLI.
2. Follow the instructions in `README.md` to set up FlexGen.

C.4 Evaluation and expected results

The raw data used for the figures in Chapter 5 can be found in `output/` directory. The scripts in `output/scripts` can be used the generate the figures in PDF format.