

Mitigating the Impact of Data Movement in Memory-Intensive Applications

Aditya Katapadi Kamath

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2026

Reading Committee:

Simon Peter, Chair

Arvind Krishnamurthy

Marco Canini

Program Authorized to Offer Degree:

Computer Science & Engineering

©Copyright 2026

Aditya Katapadi Kamath

University of Washington

Abstract

Mitigating the Impact of Data Movement
in Memory-Intensive Applications

Aditya Katapadi Kamath

Chair of the Supervisory Committee:

Simon Peter

Paul G. Allen School of Computer Science & Engineering

Modern memory-intensive workloads move terabytes of data through hierarchies of interconnects whose bandwidth and latencies have failed to scale with compute. The result is that there is no longer a singular memory wall between compute and memory, but layers of connections, each a potential bottleneck. Caches, memory bus width, and inter-device interconnects all have an impact on the performance that applications observe.

This dissertation posits that overcoming these limitations requires identifying key application characteristics that enable bottleneck mitigation. We identify and organize mitigations into three types: (1) *avoiding* data movement across certain layers by eliding extraneous or redundant data movement, (2) *hiding* data movement by overlapping transfers with complementary work, and (3) *reducing* the amount of data moved by exploiting structure present within the data itself. This characterization identifies the application properties that enable intervention, while directly informing us on how the mitigation should be designed.

We discuss three systems— $(MC)^2$, POD-Attention, and Invariant Bit Packing (IBP)—that demonstrate these concepts. Each system targets a different bottleneck component, taking advantage of application properties that enable the mitigation. They provide simple integration while retaining application correctness.

$(MC)^2$ targets bottlenecks with CPU memory access latencies, exploiting the observation

that many workloads copy data that they never fully access. It extends the CPU memory controller to lazily track requested copies. Laziness allows us to avoid copying data that is not accessed, while implementation at the memory controller targets the largest source of copy overhead—stalls on cache misses.

POD-Attention targets the GPU memory bandwidth bottleneck in LLM inference, taking advantage of the complementary resource profiles of the two phases of LLM inference, prefill and decode. It fuses compute-bound prefill and memory-bound decode so that they execute concurrently—the prefill operations occupy the compute units while decode operations saturate the memory pipeline, hiding memory stalls behind concurrent compute.

IBP targets the PCIe interconnect bandwidth bottleneck between CPU and GPU memory that occurs in large-scale ML frameworks, taking advantage of low-entropy bit patterns present across ML tensors. It strips these bits before PCIe transfer and reconstructs the original tensors within the GPU, reducing data transfer volume without loss of information.

These systems achieve substantial benefits across a wide variety of workloads. $(MC)^2$ targets databases and operating system primitives showing up to 78% higher throughput for database transactions and $250\times$ lower latency on huge page copy-on-write faults. POD-Attention enhances LLM inference providing up to 59% faster LLM attention operations. IBP improves ML offloading, providing on average 74% faster GNN training, 180% faster DLRM embedding lookups, and 24% faster offloaded LLM inference.

TABLE OF CONTENTS

	Page
List of Figures	v
List of Tables	vii
Glossary	x
Chapter 1: Introduction	1
1.1 Thesis statement	5
1.2 Summary of contributions	6
1.3 Published works	7
1.4 Outline	8
Chapter 2: Background and motivation	9
2.1 The rise of compute	9
2.2 The throughput chasm	9
2.3 Existing approaches	11
2.3.1 Avoiding	11
2.3.2 Hiding	11
2.3.3 Reducing	12
2.4 Conclusion	12
Chapter 3: $(MC)^2$: Lazy MemCopy at the Memory Controller	13
3.1 Motivation	14
3.2 Background	17
3.2.1 CPU architecture	17
3.2.2 How copies are performed	18
3.2.3 Related work	18

3.3	$(MC)^2$: Lazy Memcopy at the Memory Controller	19
3.3.1	Memory controller design	20
3.3.1.1	Copy tracking table (CTT)	20
3.3.2	Bounce pending queue (BPQ)	22
3.3.3	$(MC)^2$ functionality	22
3.3.3.1	Memory accesses	23
3.3.4	ISA design	25
3.3.5	Software design	26
3.3.6	Data protection and correctness	28
3.4	Evaluation methodology	30
3.5	Evaluation	31
3.5.1	Copy latency	32
3.5.2	Data access latency	33
3.5.3	Application workload evaluation	34
3.6	Performance tradeoff discussion	40
3.7	Conclusion	41
Chapter 4:	POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference	42
4.1	Motivation	45
4.2	Background and related work	46
4.2.1	GPU execution model	46
4.2.2	LLM inference	47
4.2.3	Related work	48
4.3	Concurrent GPU execution	50
4.3.1	Analysis of attention fusion	52
4.3.2	Analysis of fusion using a microbenchmark	53
4.4	Design of POD-Attention	54
4.4.1	SM-aware CTA scheduling	55
4.4.2	Implementing CTA-parallel fusion	56
4.4.3	Attention optimizations	57
4.4.4	Discussion on alternative implementations	59
4.5	Evaluation	60

4.5.1	Evaluating attention computation	61
4.5.2	Evaluating performance in LLM inference	62
4.5.3	Sensitivity studies	65
4.6	Conclusion	68
Chapter 5:	Reducing the GPU Memory Bottleneck with Lossless Compression for ML	69
5.1	Background and related work	71
5.1.1	PCIe transfers in ML	71
5.1.2	CPU-GPU data transfers	73
5.1.3	Related work	74
5.2	ML compression challenges	75
5.2.1	Lossy compression risks model accuracy loss	75
5.2.2	Lossless compression has high decompression overhead	76
5.3	Towards practical ML data compression	78
5.3.1	Invariance in ML data	79
5.3.2	Limits of conventional bit packing	79
5.3.3	ML data compresses well <i>across</i> tensors	80
5.4	Invariant bit packing (IBP)	81
5.4.1	Implementation of IBP	82
5.4.2	Efficient PCIe data transfer	88
5.5	Integrating IBP into ML frameworks	91
5.6	Evaluation	92
5.6.1	IBP compression performance	93
5.6.2	GNN training speedup	96
5.6.3	DLRM inference speedup	99
5.6.4	LLM inference speedup	101
5.6.5	Performance sensitivity studies	103
5.7	Conclusion	106
Chapter 6:	Future Outlook	107
6.1	Coordinated cross-layer copying	107
6.2	Automatic GPU co-scheduling	109
6.3	Pervasive lossless compression	110

Chapter 7: Conclusion	112
7.1 Concluding remarks	112
7.2 Broader implications	113
Bibliography	114

LIST OF FIGURES

Figure Number	Page
2.1 Improvement in GPU performance across generations [1, 2, 3, 4]	10
3.1 Many uses of copying have significant overhead. © 2024 IEEE	15
3.2 Source of copy overhead for Protobuf. © 2024 IEEE	16
3.3 Design overview of $(MC)^2$. © 2024 IEEE	20
3.4 Lazy copy operation in $(MC)^2$. © 2024 IEEE	23
3.5 Lazy copy on destination buffer read. © 2024 IEEE	24
3.6 Lazy memcpy function pseudocode.	27
3.7 $(MC)^2$ simplified state transition diagram.	28
3.8 Copy latency for native <code>memcpy</code> , <code>zIO</code> and $(MC)^2$. © 2024 IEEE	33
3.9 Normalized runtime for sequential accesses of destination buffer. © 2024 IEEE	34
3.10 Normalized runtime for random accesses of destination buffer. © 2024 IEEE	35
3.11 Runtime of Protobuf. © 2024 IEEE	35
3.12 MongoDB average insertion latency. © 2024 IEEE	36
3.13 Read-modify-write MVCC database throughput. © 2024 IEEE	37
3.14 Write-only MVCC database throughput. © 2024 IEEE	38
3.15 Write latencies with huge page copy-on-write. © 2024 IEEE	39
4.1 State-of-the-art FlashAttention kernels utilize either compute or memory (Model: Llama-3-8B on 2 A100 GPUs).	43
4.2 Contribution of different operations in iteration runtime with hybrid batching (model: Llama-3-8B, batch size: 60, chunk size: 1K).	45
4.3 GPU execution model.	47
4.4 Per layer attention runtime of 32 hybrid batches corresponding to chunked prefills of a request of 16K tokens (chunk size: 512, model: Yi-6B, <code>d_bs</code> : decode batch size).	52
4.5 Fine-grained fusion versus serial computation.	54
4.6 SM-aware CTA scheduling.	56

4.7	Impact of decode tile size on compute and bandwidth utilization for batch sizes 8, 16 and 32.	57
4.8	Distribution of speedup in attention computation with different mechanisms compared to FA_Serial.	61
4.9	Serving throughput in offline inference.	63
4.10	POD-Attention with varying CTA configs.	65
4.11	Effect of scheduling policy in POD-Attention.	67
4.12	Request processing throughput under varying workload distribution (model: Llama-3-8B, TP-2).	68
5.1	Conventional bit packing (left) and invariant bit packing (right). Red bits are removed to generate the compressed data shown in blue.	79
5.2	Invariant bit packing example ($T = 4$, $N = 5$).	83
5.3	Pseudocode for IBP tensor compression.	85
5.4	IBP tensor decompression steps.	87
5.5	CPU-to-GPU copy throughput across methods.	89
5.6	Integrating IBP into ML applications.	90
5.7	Decompression throughput versus space savings.	95
5.8	Average GNN training epoch speedup.	98
5.9	Breakdown of latency for GNN training epochs.	99
5.10	Normalized DLRM embedding lookup throughput varying batch and entry sizes.	100
5.11	LLM inference latency with FlexGen weight offloading.	101
5.12	Normalized LLM inference latency with InfiniGen KV cache offloading. . . .	103
5.13	Space saved with different chunk sizes and invariant threshold fractions T/N for $N = 2.4\text{M}$ tensors in Products.	104
5.14	Clustered compression net space savings.	105
5.15	LLM inference latency with varying CPU threads.	106

LIST OF TABLES

Table Number	Page
3.1 Simulated configuration. © 2024 IEEE	31
4.1 Methods of concurrently executing or fusing operations along different levels of the GPU hierarchy (GC=guarantees op co-location, WQ=reduces wave quantization).	51
4.2 Internal workload. Latency numbers in seconds.	64
4.3 arXiv-based workload. Latency numbers in seconds.	64
4.4 Per-layer attention runtime (ms) of last four prefill chunks of a prompt, co-running with decode batch size 64 (model: Llama-3-8B, context length: 16K, chunk size: 512).	66
5.1 Average speedup of compressed CPU-to-GPU transfers (best in bold). . . .	77
5.2 Average space savings (%) across algorithms/data.	77
5.3 Compression time (ms) across algorithms/data.	78
5.4 Compression of bit packing compared to the fraction of 80p-invariant bits across datasets.	78
5.5 Compression ratios with different fractions of dataset sampled during preprocessing.	95
5.6 Datasets used for GNN training. *SU are scaled up.	96
5.7 GNN tensor cache capacity with/out compression.	98
6.1 Copy coordinator decision table	108

ACKNOWLEDGMENTS

I would first like to thank my advisor Simon Peter for all his support during my Ph.D. He always provided the perfect level of guidance that I've needed: allowing me to make mistakes and explore topics on my own, while also reining me in when I get too lost in the weeds. Simon always encouraged me and fostered my interest in research. Thanks to his guidance, my Ph.D. has been one of the most rewarding experiences of my life.

I would like to thank my various collaborators: Ramya Prabhu, Ashish Panwar, Marco Canini, and Arvind Krishnamurthy, among others. Our meetings and brainstorming chats have truly accelerated the research process, while also keeping things enjoyable and light-hearted. I couldn't have asked for a better group of people to work with.

I also appreciate the various lab members whose interactions have been crucial to my development as a researcher. I'd like to specifically mention Jonggyu Park, Rajath Shashidhara, and Deukyeon Hwang as people who have given me a great deal of mentorship and advice on my Ph.D. path.

I am grateful to all the friends I have made during my journey—too numerous to name—without whom I certainly would've lost my sanity. Thank you all for bearing with my rants and introspective conversations.

A heartfelt thank-you goes to Arkaprava Basu, who showed me how rewarding research could be, and had faith that I was capable of following this path. The members of his CSL lab at the Indian Institute of Science felt like family, and I always look forward to our research chats. I would also like to thank Basavaraj Talawar, my undergraduate advisor, who pointed me towards research. I would never have found myself here without this start.

Finally, I'd like to thank my parents, who encouraged me to fully explore my passions.

My mother taught me computer programming at a young age, which sparked a lifelong interest in building computer systems. My father always saw my potential and believed in me. Not everyone has the freedom to explore their passion, and for that I feel very grateful.

They say a Ph.D. is a difficult thing to do, but the people I have supporting me have made it the easiest thing in my life.

Disclosure: Generative AI tools, including Claude [5] and ChatGPT [6], were used in the writing process for editing and revising prose, and for feedback on structure, argumentation, and clarity of exposition. All research contributions, including system designs, experiments, and conclusions, are my own.

GLOSSARY

Compute

CPU (*Central Processing Unit*): The main processor in a system. Runs the operating system and distributes work to other accelerators.

ISA (*Instruction Set Architecture*): The set of instructions a processor can execute, defining the interface between software and hardware.

RFO (*Read-For-Ownership*): A cache coherence operation that reads a cacheline with the intent to modify it, acquiring exclusive ownership of the line.

ROB (*Reorder Buffer*): CPU hardware component that enables out-of-order execution by holding instructions that are still in-flight. Instructions in the ROB are completed and removed in program order.

FLOPS (*Floating point Operations Per Second*): A unit of throughput, typically used to indicate the maximum processor throughput.

GPU architecture

GPU (*Graphics Processing Unit*): A specialized processor designed for parallel computation. Originally for graphics rendering, but now widely used for general-purpose parallel workloads such as AI or scientific simulations.

Tensor Core: Specialized cores within a GPU that perform matrix-matrix multiplications with very high throughput.

SM (*Streaming Multiprocessor*): The basic deployment unit of an NVIDIA GPU, containing tensor cores, integer and floating-point execution units, a register file, an L1 cache, and shared memory. Modern GPUs contain around a hundred SMs.

Warp: A group of 32 GPU threads that execute in lockstep within an SM.

CTA (*Cooperative Thread Array*): A group of GPU warps that share the L1 cache and shared memory. All warps in a CTA execute within a single SM.

Kernel: A function executed on the GPU, launched from the host CPU with a specified number of CTAs and shared memory. Not to be confused with OS kernel. In this document, kernel usually refers to a GPU kernel.

UVM (*Unified Virtual Memory*): A CUDA feature providing a single address space across CPU and GPU memory, letting the GPU kernel access CPU memory directly, without the CPU's involvement.

Coalesced memory access: A GPU memory access where neighboring threads in a warp access contiguous memory addresses, allowing the hardware to combine them into a single wider transaction.

Wave: A set of CTAs running concurrently on the GPU.

Wave quantization: Underutilization that occurs when the number of deployed CTAs is not a multiple of the number the SMs can hold concurrently, leaving some SMs idle in the final wave.

Memory

DRAM (*Dynamic Random Access Memory*): A type of volatile memory used as main memory in most computer systems.

SRAM (*Static Random Access Memory*): A faster, more expensive type of volatile memory, typically used for caches.

HBM (*High Bandwidth Memory*): A stacked DRAM technology providing high bandwidth and low power consumption, commonly used as GPU memory.

DMA (*Direct Memory Access*): A feature that allows hardware (e.g., GPU or disk) to transfer data between memory regions without involving the CPU.

DMA engine: A hardware component that uses DMA to copy data between memory regions without involving the CPU.

Cacheline: The unit of data transfer between a cache and main memory, typically 64 bytes for x86 CPUs and 128 bytes for NVIDIA GPUs.

LLC (*Last-Level Cache*): The cache level closest to main memory, typically shared across all CPU cores.

TLB (*Translation Lookaside Buffer*): A memory cache that stores recent virtual-to-physical address translations to accelerate memory access.

Huge Page: A larger-than-default memory page (e.g., 2MB instead of 4KB on x86) that reduces TLB misses for applications with large working sets.

COW (*Copy-on-Write*): A memory management technique where a page is shared between processes until one of them writes to it, at which point a copy is made.

RMW (*Read-Modify-Write*): A memory operation that reads a value, modifies it, and writes the result back as a single logical operation.

MC (*Memory Controller*): The hardware component that manages the DRAM, transferring data to/from the CPU.

Memcpy (*Memory Copy*): A standard library function that copies a block of memory from a source to a destination buffer, where the buffers do not overlap. Depending on context, it may also generically refer to copies between two non-overlapping buffers, e.g., OS memcpy.

Interconnects

PCIe (*Peripheral Component Interconnect Express*): A high-speed serial bus standard that connects peripheral devices such as GPUs and storage devices to the CPU.

NVLink: A high-bandwidth interconnect by NVIDIA for GPU-to-GPU communication.

CXL (*Compute Express Link*): An open standard interconnect for cache-coherent CPU-to-device and CPU-to-memory connections.

InfiniBand: A high-bandwidth interconnect used for networking within a data center.

Databases

MVCC (*Multi-Version Concurrency Control*): A database transaction technique that maintains multiple versions of data to allow concurrent reads and writes without locking.

Tuple: A row in a relational database table.

OLTP (*Online Transaction Processing*): A workload pattern of concurrent transactions on a database, typical of business applications like order processing or inventory management. Consists of reading or modifying data.

TPC-C: An industry-standard benchmark for OLTP systems.

Machine learning (ML)

LLM (*Large Language Model*): A neural network trained on large text corpora to process natural language.

GNN (*Graph Neural Network*): A class of neural networks that operate on graph-structured data, learning representations of nodes and edges.

DLRM (*Deep Learning Recommendation Model*): A class of neural networks used for predicting user preferences, characterized by large embedding tables for categorical features.

Tensor: A multi-dimensional array, a fundamental data unit in ML frameworks.

Token: A discrete unit of text, typically a word or sub-word, represented as a numerical ID processed by the LLM. In some contexts it also refers to the unit of tensors that the LLM operates on.

Tokenization: The process of converting text into a sequence of numerical tokens for input to a language model.

Attention: A core operation in transformer-based models that computes the relative importance of each token with respect to every other token.

Prefill: The first phase of LLM inference, where the model processes the input prompt to produce the first output token. Compute-bound.

Decode: The second phase of LLM inference, where the model generates output tokens one at a time. Memory-bound.

KV cache (*Key-Value Cache*): The storage of intermediate key and value tensors computed during LLM attention, reused across decode iterations to avoid recomputation.

QSL (*Query Sequence Length*): The number of input query tokens in a request processed during attention; small for decode (1–8), large for prefill.

GQA (*Group Query Attention*): An attention variant where each key/value head is shared across a group of query heads, reducing KV cache size relative to earlier multi-head attention.

TTFT (*Time To First Token*): The latency to produce the first output token of an LLM request i.e., the prefill latency.

TBT (*Time Between Tokens*): The latency between successive output tokens during decode.

FFN (*Feed-Forward Network*): The linear (matrix-matrix) layers of a transformer block, distinct from attention.

Hybrid Batching: A scheduling technique that combines prefill and decode requests in the same batch to share model weights and improve overall throughput.

Embedding Table: A lookup table that maps categorical inputs to dense tensor representations, used in recommendation models.

Offloading: A technique that stores ML model data (weights, activations, KV cache) in larger CPU memory and transfers it to the GPU on demand.

Compression

Lossless Compression: Compression where the exact original data can be reconstructed.

Lossy Compression: Compression that achieves higher compression ratios by allowing some loss of information.

Quantization: A lossy compression technique that reduces the precision of numerical values to reduce memory footprint, data movement, and/or computation.

Bit Packing: A compression technique that removes redundant bits (e.g., leading zeros) from data items to reduce their storage size.

Space Savings: The percentage reduction in data size achieved by compression. Higher is better.

$$\text{Space savings (\%)} = \left(1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}} \right) \times 100$$

Compression Ratio: The ratio of uncompressed data size to compressed data size. Higher is better.

$$\text{Compression ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}$$

Thesis-specific terms

(MC)² (*Memory Copy at the Memory Controller*): The lazy copy system presented in Chapter 3.

POD-Attention: The fused prefill-decode attention kernel presented in Chapter 4.

IBP (*Invariant Bit Packing*): The lossless compression scheme presented in Chapter 5.

CTT (*Copy Tracking Table*): The SRAM structure added to the memory controller in (MC)² to track prospective copies.

BPQ (*Bounce Pending Queue*): The structure added to the memory controller in (MC)² to hold writes to source buffers during lazy copy execution.

MCLAZY: The ISA instruction introduced by (MC)² to issue a lazy memory copy.

MCFREE: The ISA instruction introduced by (MC)² to hint that a destination buffer is no longer in use.

SM-aware CTA scheduling: The in-GPU scheduler that decides whether to perform prefill or decode during kernel runtime based on the SM that the CTA was scheduled onto.

Chapter 1

INTRODUCTION

Modern applications—like databases or machine learning—process data at the scale of hundreds of gigabytes to terabytes, moving this data through a vast hierarchy of interconnects and buffers including caches, on-chip networks, and PCIe interconnects. Despite computational throughput tripling roughly every two years, the bandwidth and latency of the memory and interconnects supporting it has not kept pace [7, 8]. As a result, data movement, and not computational throughput, often determines application performance. Cache misses, execution serialization, and resource interference all exacerbate bottlenecks in the hierarchy, inducing latency spikes, throughput reductions, and stalls on the critical path.

The bottleneck layer in the hierarchy dictates the performance that applications observe. DRAM bandwidth and latency limitations cause copy-heavy workloads to stall waiting for reads. For example, in databases using multi-version concurrency control (MVCC) for transactional isolation [9, 10], copy operations consume up to 68% of CPU cycles for write-heavy transactions due to stalls [11]. Similarly, GPU memory bandwidth limitations restrict throughput for long-context Large Language Model (LLM) inference. Here, over 60% of time is spent on attention operations [12, 13]. This is split between a compute-bound phase that leaves memory bandwidth largely idle, followed by a memory-bound phase that leaves compute idle. Limitations in PCIe bandwidth cause problems for offloading-based ML systems [14, 15, 16], which keep data that exceeds GPU memory capacity within CPU memory. PCIe transfers from CPU memory can consume up to 70% of execution time for offloading [17, 18, 15], where the GPU is stalled waiting for data to arrive.

This dissertation prescribes three complementary techniques that address bottlenecks across the hierarchy. This taxonomy both identifies application properties that enable mit-

igation and informs us of how to take advantage of them. We *avoid* data movement when it is unnecessary, *hide* movement overhead by overlapping with complementary work, and *reduce* the amount of data moved by taking advantage of structure within the data itself. We present three systems: $(MC)^2$ [11], POD-Attention [12], and IBP [17], that apply these techniques to different bottlenecks, achieving substantial speedups for databases, operating system primitives, and machine learning without mandating application redesign.

Avoiding: This technique requires identifying unnecessary data movement within applications, then providing an alternate path that retains the properties provided by the movement without its overhead. For example, in MVCC databases, transactional isolation is guaranteed by copying the tuple(s) for each transaction intending to modify it [10]. This gives each write transaction a private buffer that it can modify as needed without risk of interference from concurrent transactions. However, these buffers are frequently only partially accessed, as transactions modify only a subset of the tuple, but the entire tuple must still be copied for correctness. We observe similar patterns in serialization libraries, IO buffers, and copy-on-write fault handling, where copying is used for isolation, and not because the copied buffer is actually consumed. Simply removing these copies would harm application correctness. We find that the primary source of overhead during copying is cache misses. Cached data is fetched and moved quickly, while uncached data requires accessing memory, which is over an order of magnitude slower. These memory access instructions fill the CPU’s reorder buffer leading to stalled cycles, where the CPU is stuck waiting for accesses to complete.

To avoid copy overheads while preserving its isolation benefits, we propose lazily copying at the memory controller, i.e., $(MC)^2$. $(MC)^2$ leverages the insight that the memory controller observes every read operation that misses the cache, allowing it to track and defer copies until data is actually read. Rather than eagerly copying, $(MC)^2$ simply tracks the source-destination pair. If the destination is read or source written the copy is performed, otherwise it is elided. We guarantee that the CPU always sees copied data correctly, preserving application semantics while only paying the copy overhead for used data. $(MC)^2$ operates

at a cacheline granularity, speeding up copies on the order of hundreds of bytes, prevalent across workloads. $(MC)^2$ achieves up to 78% higher throughput for MVCC databases, 43% faster Protobuf processing, and $250\times$ lower latency for huge page copy-on-write faults.

Hiding: For this technique, we must identify complementary work that can be done alongside the data movement, hiding the movement overhead in the critical path. LLM inference consists of two phases with complementary resource usage: a compute-bound prefill phase that processes the input prompt, followed by a memory-bound decode phase that generates output tokens one at a time [19]. Prefill performs large matrix-matrix operations, saturating GPU compute while leaving memory bandwidth mostly idle. Decode operates on a single token, performing memory-bound matrix-vector operations that demand high bandwidth while leaving compute mostly idle. Existing systems batch prefill and decode together for linear operations (feedforward network), but compute attention serially for each phase using functions optimized for either compute or bandwidth utilization. As context lengths grow, attention overheads dominate, with 60% of the inference time being attention at 16K tokens.

By overlapping the two operations, we improve both the compute and memory bandwidth utilization—hiding the memory stalls of decode attention behind concurrent compute of prefill attention. The critical challenge is managing co-location: for peak speedup, both operations must run within the same GPU streaming multiprocessor (SM), since each SM has its own compute units and memory pipeline. With POD-Attention, we guarantee co-location through SM-aware CTA scheduling. When groups of threads launch, a hardware register is read to determine which SM they were scheduled on. They then dynamically decide whether to perform prefill or decode based on what is already running on the same SM. POD-Attention provides up to 59% faster LLM attention and improves end-to-end LLM inference throughput by up to 22% over state-of-the-art.

Reducing: This technique involves identifying patterns within data being moved, then exploiting them to reduce the volume of data transferred. Machine learning frameworks fre-

quently operate on datasets that exceed GPU memory capacity, e.g., graph neural networks (GNNs) [20, 21] operate on terabyte-scale graphs, recommendation models (DLRM) deal with billions of embeddings [22], and LLMs operate over million-token contexts [23]. ML frameworks address this capacity limitation with offloading, where the significantly larger CPU memory is used to store data, transferring to the GPU on demand. This shifts the bottleneck from GPU memory capacity to PCIe interconnect bandwidth, as every access to offloaded data must traverse the interconnect.

Many of these frameworks already rely on *avoiding* and *hiding* data movement with caching and parallel processing respectively [16, 23]. Frameworks maintain a software cache in GPU memory for frequently accessed tensors, avoiding repeatedly fetching them from across the PCIe. They overlap the PCIe fetches by computing on the tensors already in GPU memory. Despite using these mitigations, we find that GPU compute still stalls while waiting for transfers, consuming up to 70% of execution time [17].

A natural solution is to reduce the amount of data being transferred. Compression is a promising candidate—transferring compressed data to the GPU then decompressing before use. Lossy compression comes with risks of information loss and unpredictable behavior [24], and so, we restrict our view to lossless compression, where no accuracy risks arise. We find that existing lossless algorithms are slower than transferring the original data, as the decompression overhead and metadata sizes outweigh any benefits in traffic reduction. An ideal algorithm would minimize compression metadata while keeping decompression cheap. For this, we analyze ML datasets to identify features that allow us to provide this decompression.

We observe that ML data frequently has low-entropy bit patterns that repeat *across* tensors. The exponent and sign bits of floating-point values often have similar values when drawn from the same dataset. IBP identifies these *invariant* bits, stores them once in a small metadata structure (on the order of kilobytes) within the GPU, then strips them from the tensors. Decompression uses cheap bit shift operations coupled with asynchronous memory accesses, eschewing dictionary lookups or heavyweight code tables used by conventional compression. IBP accelerates GNN training by 74%, DLRM embedding lookups by 180%,

and LLM inference by 24% on average, while avoiding any loss of information.

Composability of techniques: Classifying data movement mitigations into avoiding, hiding, and reducing provides a clear link between application properties and mitigation strategies. The framework is composable rather than divisive—systems are not constrained to a single technique. While each system in this thesis is dominated by one technique, they also draw on others wherever applicable. $(MC)^2$'s primary technique is *avoiding*, eliding redundant copies whose destinations are never accessed. It also performs *hiding*, as the memory controller overlaps data copying with CPU computation. POD-Attention's primary technique is *hiding*—overlapping LLM phases with complementary resource requirements. By improving GPU utilization, POD-Attention also allowed us to reduce data replication other attention kernels use to increase GPU utilization [25], consequently *avoiding* the extra memory accesses needed for replication. Finally, IBP's primary technique is *reducing*, but it also uses asynchronous memory accesses to *hide* the overhead of decompression.

Our taxonomy provides a lens to spot performance opportunities, instead of partitioning systems into disjoint categories. High-performance systems should apply multiple techniques to address bottlenecks across layers.

1.1 Thesis statement

Avoiding, hiding, and reducing data movement provides a clear framework for mitigating the impact of data movement in memory-intensive applications, targeting bottleneck layers in the memory hierarchy through mitigations that exploit specific application characteristics. Data movement should be *avoided* when applications request it unnecessarily. When data movement cannot be avoided, its overhead should be *hidden* by overlapping with other useful work. Finally, the amount of data moved should be *reduced* when the structure of data permits it.

This thesis presents three systems that apply these techniques to different hardware components, achieving latency reduction and throughput improvement. $(MC)^2$ extends the

memory controller, tracking copies and deferring them until data is actually accessed. It avoids the copy entirely when destinations are not accessed, accelerating MVCC databases, interprocess communication, and operating system primitives. POD-Attention operates on the GPU streaming multiprocessor (SM), overlapping compute-bound and memory-bound phases of LLM inference through SM-aware scheduling that guarantees fine-grained co-location. IBP targets the PCIe interconnect, losslessly compressing ML tensors by taking advantage of low-entropy, invariant bits present within them. Decompression within the GPU uses cheap bit shift operations and asynchronous memory accesses, accelerating GNN training, DLRM table lookups, and LLM inference. All three systems— $(MC)^2$, POD-Attention, and IBP—are designed to integrate well with existing applications, minimizing the code changes required for deployment.

1.2 Summary of contributions

We make the following contributions:

- We analyze data movement bottlenecks present across multiple domains: memcpy consumes up to 68% of cycles in serialization-heavy workloads, causing the CPU to be stalled by transfers from memory. PCIe transfers consume up to 70% of ML offloading execution time, causing the GPU to stall.
- We present $(MC)^2$, a memory controller extension that supports a lazy memory copy operation. It ensures data appears as copied to the CPU, guaranteeing correctness, but avoids the actual copy when data isn't accessed. Placing this mechanism at the memory controller allows it to eliminate the major source of overhead in memcpy operations, the memory access overhead of cache misses. $(MC)^2$ achieves 43% faster Protobuf processing, up to 78% higher database throughput, and $250\times$ lower latency for huge-page copy-on-write faults.
- We demonstrate POD-Attention, a hybrid GPU attention kernel that combines compute-

heavy prefill and bandwidth-bound decode requests into a single kernel. Both operations execute concurrently within the same GPU streaming multiprocessor. This allows POD-Attention to hide memory stalls from decode behind concurrent prefill compute, accelerating attention by up to 59%, and improving end-to-end inference throughput by up to 22% over the state-of-the-art.

- We present IBP, a lossless GPU-optimized compression algorithm, that exploits low-entropy, invariant bits present across ML tensors, such as the exponent bits of floating-point data. IBP stores these bits in a small metadata structure within the GPU memory (on the order of kilobytes), and removes them to compress the tensors, reducing the amount of data moved. IBP accelerates GNN training by 74%, DLRM embedding lookups by 180%, and LLM inference by 24% on average.

All of our code is open source:

(MC)²: <https://github.com/AKKamath/MCSquare-ISCA24>

POD-Attention: https://github.com/microsoft/vattention/tree/main/pod_attn

IBP: <https://github.com/AKKamath/InvariantBitPacking>

POD-Attention has also been integrated into the open-source library FlashInfer [26] and is being integrated into the SGLang [27] framework. Both are widely used in production LLM deployments, and the work has garnered significant interest from industry.

1.3 Published works

- Aditya K Kamath, Simon Peter. 2024. **(MC)²: Lazy MemCopy at the Memory Controller**. In 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA '24) DOI: 10.1109/ISCA59077.2024.00084
- Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, Ashish Panwar. 2025. **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 (ASPLOS ‘25) DOI: 10.1145/3676641.3715996

- Aditya K Kamath, Arvind Krishnamurthy, Marco Canini, Simon Peter. 2026. **Reducing the GPU Memory Bottleneck with Lossless Compression for ML**. In Proceedings of the 21st ACM European Conference on Computer Systems (EuroSys ‘26) DOI: 10.1145/3767295.3803595

1.4 Outline

This thesis is laid out as follows. Chapter 2 examines how compute has vastly outpaced the latency and bandwidth of other interfaces, such as memory and interconnects, and provides a background on work done to manage this scaling. Chapter 3 then discusses $(MC)^2$ and how it avoids data movement for database and operating systems. Chapter 4 discusses POD-Attention, which hides data movement by overlapping compute-heavy prefill with bandwidth-heavy decode to improve LLM throughput and reduce inference latency. Chapter 5 goes over IBP, which reduces the amount of data transferred across the PCIe, improving offloading-based machine learning workloads like GNNs, DLRMs, and LLMs. Chapter 6 then discusses future directions for exploration. Finally, Chapter 7 concludes this thesis.

Chapter 2

BACKGROUND AND MOTIVATION

2.1 *The rise of compute*

Historically, computational throughput has been the dominant metric for improvement in computer hardware [28, 29]. In the 1990s and early 2000s, gains were largely driven by increased frequency scaling. After Dennard scaling [30] ended and multi-core architectures emerged, parallelism and scale-out became the source of compute growth. Newer architectures like GPUs and TPUs delivered orders of magnitude more compute through simplified execution units and vast parallelism, enabling applications like machine learning or scientific simulations to become tractable at scale [31, 32, 33].

However, this historic growth in compute comes with a widening gap between processor throughput and the rate at which data can be moved through the system. While compute throughput scales with transistor density, data movement is constrained by other factors. Physical distance between the compute and memory alongside bus widths impact the rate at which data can be moved. These constraints scale slowly, leaving structural limitations that persist across generations.

2.2 *The throughput chasm*

The term *Memory Wall* was coined in 1994 to denote the gap between CPU compute and DRAM [34]. The term has since become ubiquitous in descriptions of data movement bottlenecks. This wall has not narrowed in the decades since—instead, multiple walls have emerged across the layers of a system.

To concretize the gaps between compute, memory, and interconnects, we examine the performance of server-class NVIDIA GPUs and supporting components across generations

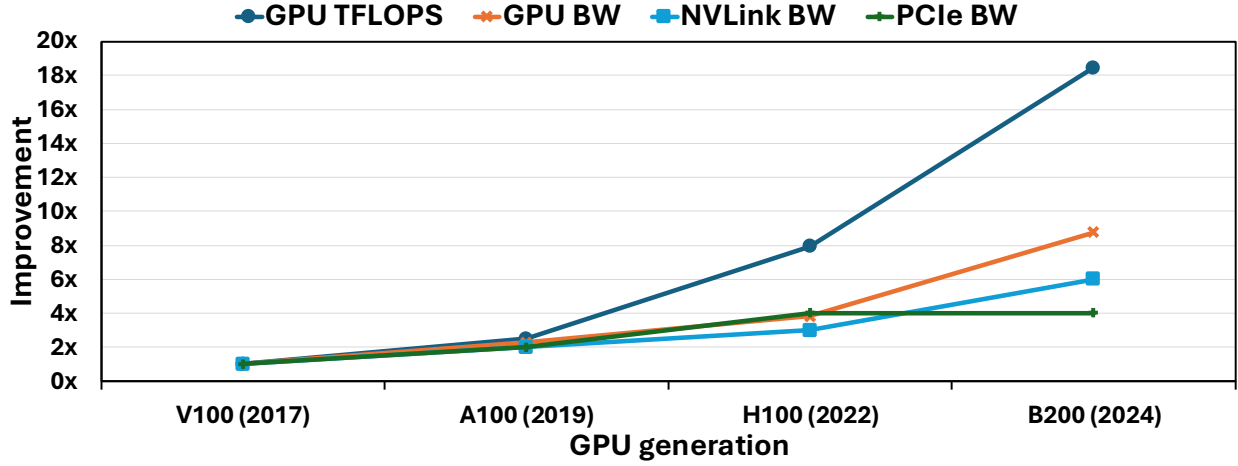


Figure 2.1: Improvement in GPU performance across generations [1, 2, 3, 4]

shown in Figure 2.1. From V100 (2017) to the latest B200 (2024), GPU compute throughput has increased by 18 $\times$ over seven years, while on-package GPU memory bandwidth has increased only 9 $\times$. Inter-device interconnects have scaled even more slowly: NVLink, the dominant GPU-to-GPU interconnect, has improved by 6 $\times$, and PCIe, the standard CPU-GPU interconnect, by only 4 $\times$. Similar trends have been observed with CPUs as well [8, 7].

The result is a layered hierarchy, with each successive layer scaling slower than the layer above it. Compute scales the fastest, while memory bandwidth lags. Interconnects compound the problem, scaling slower than both. As workload sizes grow, the bottleneck moves to the slowest layer that the data has to traverse. The same compute that enables processing larger workloads also causes different layers of the hierarchy to manifest as performance constraints.

Different workloads interact with this hierarchy differently. This thesis looks at three specific bottleneck components for multiple workloads: CPU memory, GPU memory, and the PCIe interconnect. For each, we design techniques to address the bottleneck for that layer, exploiting application characteristics to enable the mitigation.

2.3 Existing approaches

Data movement mitigations have been heavily explored by prior literature. We now briefly go over how prior systems have made use of avoiding, hiding and reducing data movement.

2.3.1 Avoiding

Avoiding data movement requires identifying movement that isn't strictly necessary, and providing alternative methods that preserve the guarantees provided (e.g., data isolation). Zero-copy frameworks provide APIs that reduce or eliminate copies between the OS and application memory [35, 36, 37, 38]. Making use of them typically requires rewriting application code to call the new APIs. Operating systems like Linux use copy-on-write [39] during *fork()* to avoid copying pages until they are written to, eliminating copies for pages that remain unmodified.

Hardware [40, 41] and software [42, 20, 43] caching are popular methods of avoiding data movement: frequently accessed data is identified and kept closer to the processor, avoiding repeated transfers. Near-memory-processing (NMP) and processing-in-memory (PIM) are gaining prominence, where the compute is moved to the data instead of vice-versa, avoiding large data transfers through limited links and exploiting the high internal bandwidth of memory modules [44, 45, 46, 47].

2.3.2 Hiding

Hiding overlaps data movement with other concurrent operations (typically compute) so that transfer overhead is removed from the critical path. This typically relies on hardware support for asynchronous data movement. Prefetching loads data before it is needed, allowing the access to be overlapped with ongoing operations. This can be done both through hardware [48, 49, 50] as well as software [51, 52, 53, 54]. Asynchronous memory accesses, through hardware like Direct Memory Access engines [55], allow data copies to be initiated ahead of time, with a synchronization later in the program to ensure the copy completed [56, 57].

Hyperthreading is a form of latency hiding, where memory stalls of one thread are hidden by quickly switching execution to a different thread. In the GPU realm, streams and concurrent kernel execution are popular ways to manage operation overlap, which provides programmers with an explicit way to specify operation dependencies [58]. Compute-communication overlap is a popular form of hiding where compute is broken into batches then overlapped with network transfers [59, 60, 61].

2.3.3 Reducing

Reducing data movement involves identifying patterns in data that can be exploited to reduce the volume of data transferred. For ML, quantization reduces tensor precision (e.g., FP32 to FP8) shrinking the volume of data moved at the cost of accuracy [62, 63, 64, 65, 66]. Model pruning and sparsification [67, 68] have also been proposed to reduce the volume of data by removing low impact weights. For storage and networking, lossless compression has been deployed to reduce the amount of data moved across interconnects [69, 70]. In hardware, cache compression has been proposed to increase effective cache capacity, reducing the data that needs to be fetched from memory [71].

2.4 Conclusion

Historically, computational throughput has outpaced advances in both memory and interconnect performance. This leads to a widening gap that creates application bottlenecks at different components. Systems and architecture research has tried to manage these bottlenecks using the techniques of avoiding, hiding, and reducing data movement. Each technique takes advantage of specific workload properties that enable them.

Chapter 3

$(MC)^2$: LAZY MEMCOPY AT THE MEMORY CONTROLLER

This chapter discusses $(MC)^2$, which uses the *avoiding* technique for data movement mitigation, eliminating memory copies that applications request but do not fully access¹. We extend the memory controller to track copies lazily, deferring the copy until either the source buffer is written to or the destination buffer is read, allowing us to elide copies to cachelines that are not accessed. This addresses the bottleneck caused by cache misses during copying, where the CPU is stalled waiting for data from memory.

Data copying has a significant impact on the execution latency and overhead of modern applications. Profiling of Google’s datacenters shows that more than 5% of all CPU cycles are consumed by memory copy (memcpy / memmove) operations [72, 73], a *substantial* overhead at that scale. Additionally, waiting for copy completion adds “killer microseconds” [74] to application processing which can’t be hidden by classical means like out-of-order processing. Most CPU cycles for memcpy are spent stalled, waiting for data from memory.

Memory access latency is not expected to improve in the future due to the classic “memory wall” problem, where technological advances improve clock speeds for CPUs, while memory latencies remain largely stagnant [75]. DDR5 improves memory bandwidth by up to 2× over DDR4 at the cost of memory access latency [76]. Memory latencies are expected to worsen as cloud providers add high capacity memories at the expense of latency [77, 78, 79, 80].

The common use case of data copying is for managing temporary buffers. Temporary buffers are widely used due to the strong isolation and ownership guarantees that they provide. Data that needs to be shared by multiple components of an application can be copied into temporary buffers, giving each component its own local copy. These components can

¹This chapter contains text and figures from the original IEEE publication © 2024 IEEE [11].

then access and modify data in the buffer without risk of interference by other components. This simplifies program logic, as multiple components of an application do not need to synchronize for local buffer access. In many of these cases, the utility is so great that copies into temporary buffers are used even if not all of the copied data is modified or even accessed.

3.1 *Motivation*

To concretize our discussion, we look at some cases where data copies provide great utility.

Serialization. This is an operation where data structures are converted into a format that can be easily transferred between processes. For example, Google’s Protobuf [81] is a popular library used for language-agnostic serialization. The process of serialization involves taking a data structure and converting it into a stream of bytes. For this, a buffer is allocated and the data structure is processed and copied into the buffer. The buffer is then sent to another process, where it is deserialized, i.e., converted back into data structure form.

Many works in the field of ML [14, 82, 83] have noted that copy overheads incurred during serialization have made multiprocessing in Python infeasible. Prior work [84] proposes an accelerator for Protobuf due to high serialization/deserialization overheads, including copies.

IO buffers. Prior work [38] has noted that IO-intensive applications and the operating system IO stack often make several redundant data copies. Applications like MongoDB make use of copied buffers to pass data between independent subsystems. These subsystems can modify the data for their specific purpose without having to worry about other subsystems modifying or freeing the buffer, e.g., one subsystem may log data while another inserts it into a hash table. These copies could have been avoided by keeping track of buffer ownership, but this involves complicated, fine-grained memory management and book-keeping to ensure buffers are not freed or modified before subsystems have finished reading.

Multi-version concurrency control (MVCC). Traditional methods of isolation in databases involve the use of locks to ensure that data being modified is not simultaneously read. However, in cases of high contention, this leads to high wait times between readers and writers. To avoid this, in MVCC [10] databases, writers instead create copies of tables or tuples being

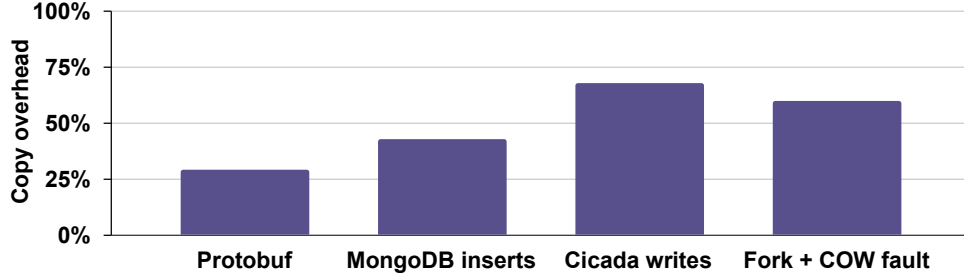


Figure 3.1: Many uses of copying have significant overhead. © 2024 IEEE

written to and only modify their local copy. At commit, these copies are integrated into the main database. Concurrent readers can read data from the main database without worrying about reading partially modified data. However, this comes at the tradeoff that writers may end up copying data that ends up not being modified.

These use cases for temporary buffers exhibit high copy overhead. Figure 3.1 shows the copy overhead for four such cases which we obtained by running applications on an Intel Skylake server, measuring CPU cycles attributed to memory copy using Linux perf [85]. We see that copy overhead in terms of cycles spent in memcpy can be up to 68%.

Protobuf is a workload from Google’s Fleetbench suite [86] that executes protocol buffer [81] operations based on traces from Google’s servers. MongoDB [87] is a popular NoSQL server that prior work [38] has shown exhibits redundant copy operations to manage IO buffers. Cicada [88] is an MVCC relational database that makes use of copies during write operations for transactional isolation. Finally, `fork` [39] is a common system call used to create a child process, which creates a copy of the parent page table, marking all pages as copy-on-write (COW). When a page is modified, a page fault is triggered and the page is copied. Figure 3.1 shows a significant portion of this page fault handling is spent on copying data for 4KB pages. For 2MB huge pages, this overhead can reach 99%.

Virtual snapshotting [89, 90] is a technique used by in-memory databases that takes advantage of this feature to take consistent snapshots of the database. This is done by

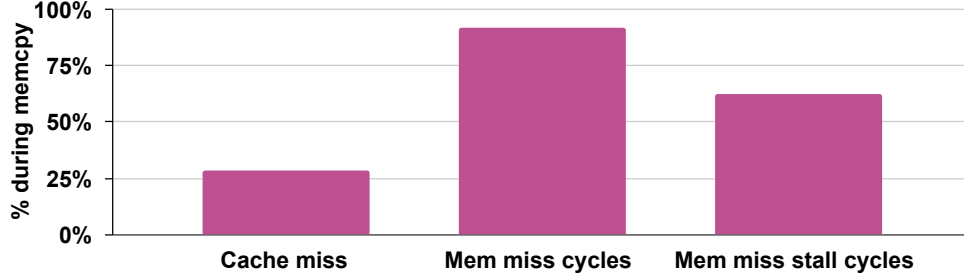


Figure 3.2: Source of copy overhead for Protobuf. © 2024 IEEE

launching a new process whenever a snapshot is needed. The new process then has a virtual copy of the entire in-memory database. While extremely useful, this technique can have high copy overheads in the critical path. For this reason, databases like Redis advocate against the use of huge pages [91], despite the performance benefits of huge pages, as the copy overhead causes unavoidable latency spikes.

To break down the sources of this copy overhead, we took a deeper look at different perf metrics when running Protobuf, shown in Figure 3.2. Around 25% of the data accesses during copying end up missing the cache. These accesses then have to retrieve the data from memory. 90% of the time, there is an instruction in the CPU waiting for a memory access (Mem miss cycles). These instructions reserve a slot in the CPU’s reorder buffer (ROB), that could block further instructions from entering. 60% of the time, the CPU is completely stalled (Mem miss stall cycles) as the ROB is full and cannot service further instructions.

This highlights the source of the copy overheads faced by applications. Copy operations issue reads and writes to move data. Data which is cached can be moved quickly, while cache misses end up composing the majority of the time that the copy operation takes. To remedy this situation, we must focus on improving movement for data being serviced from memory during copy operations.

When we analyzed these applications deeper, we found that in many of these cases only a fraction of each copy is accessed, making the remainder of the copy redundant. For example,

in Cicada [88], when an attribute of a record is modified, the entire record must be copied for transactional isolation, whereas the amount of data being modified may only be a small fraction of the record that was copied.

To alleviate this overhead, our proposal relies on lazy execution of each memory copy. Laziness is a common technique used across different domains in computer science. When an expensive operation is requested, laziness delays the operation until time of use. For example, functional languages [92] use it to allow declaration of infinitely sized data structures that consume limited memory—only upon data access are the operations resolved and memory is allocated. Operating systems use copy-on-write [39] to avoid performing copies until pages are modified. Laziness provides the advantage that penalties are paid only upon use.

Recent advances in memory and processing logic have allowed compute logic to be placed near memory modules, such as within the memory controller [44]. Prior work has taken advantage of this to integrate tasks such as encryption [93], data address remapping [94], or logging [95] within the memory controller. As all memory accesses are marshaled by the memory controller, we believe it is the ideal place to improve data copying mechanisms.

3.2 Background

In this section, we shall briefly cover how data copying is performed, as well as look at how related systems try to handle this issue.

3.2.1 CPU architecture

To understand how the memory system affects copy overhead, we briefly cover the typical hardware architecture of a memory system. This system consists of a set of CPU cores, each with their own private cache and a shared last-level cache (LLC). The system also contains a set of memory modules with memory controllers responsible for issuing operations to these modules. An interconnect connects the memory controllers to the CPU cores via the caches.

For memory read operations, the system first probes the CPU’s private cache and the shared LLC to locate the requested data. If unsuccessful, the system transmits the access via

the memory interconnect to the appropriate memory controller. This controller then sends a request to the memory module and, upon retrieval, forwards the data to the core through the caches. In contrast, write operations typically involve a direct write to the cache, with the data eventually reaching the memory through subsequent cache evictions.

In this architecture, the access latency becomes progressively worse as the distance from the CPU core increases. The typical range of memory access latencies can span up to 3 orders of magnitude, from a few nanoseconds for an L1 cache access to hundreds of nanoseconds for DRAM. Reads to the source buffer that miss in the cache have the biggest effect on memcpy overhead, stalling the CPU.

3.2.2 How copies are performed

Memory copies are used to transfer a given size of data from a source buffer to a destination buffer. These buffers do not overlap. There are four principal operations involved: load, store, test, and loop. The load fetches data from the source buffer, and the store places it into the destination buffer. The test operation checks whether the current copied size matches the provided size. If not, the loop operation restarts the process for the next iteration. SIMD instructions are leveraged if available [96, 97] until the entire buffer has been transferred. While modern CPUs support out-of-order and speculative execution which extract parallelism across iterations of copying, other work [98] has found that this is limited. As the number of copy iterations increases, the reorder buffer quickly fills, forcing further iterations to wait. This brings memory access latencies into the critical path for the copy.

3.2.3 Related work

A variety of techniques have been proposed in literature to reduce this overhead. The closest line of work proposes a memcpy accelerator within the cache [99, 100, 101, 102]. This accelerator adds a mapping table within the cache that remaps destination buffer requests to the source buffer. However, the performance of this accelerator is contingent on the source data being present in the cache. Otherwise, it is still fetched from memory. A tangential line

of work explores in-DRAM copying [45, 103, 104]. These take advantage of high-bandwidth internal links present in DRAM to perform copy operations fully within the DRAM module, without needing to move data across the memory interconnect. These proposals require the source and destination buffers to be present within the same DRAM module.

Alternative proposals present copy engines to improve bandwidth and reduce CPU data movement overhead [105, 57, 98, 106, 55, 107]. However, many of these have high initialization overhead, making them impractical for kilobyte-sized copies. We also view these proposals as addressing an orthogonal problem to ours. $(MC)^2$ avoids unnecessary data movement, removing copy overheads from the critical path.

Many proposals target copy overheads in specific application domains. Demikernel [108] adds zero-copy APIs to IO stacks, along with programming-language-based object ownership tracking features. Several works target reducing copy overheads during serialization [36, 84, 109, 35]. zIO [38] eliminates redundant memory copies present in IO-based application and OS operations. Contrary to these, we seek to provide a general-purpose solution that can be applied across different application domains.

3.3 $(MC)^2$: *Lazy Memcopy at the Memory Controller*

We propose performing Memory Copies lazily at the Memory Controller, i.e., $(MC)^2$. $(MC)^2$ augments the memory controller to allow programmers to issue a lazy memcopy operation on a source and destination buffer. When any CPU or cache writes to the source buffer or reads from the destination buffer, the memory controller performs the copy lazily. At all times, data appears to the program as if it had been copied eagerly. $(MC)^2$ allows applications to avoid the latency of copying in the critical path while only exhibiting overheads for copied data that ends up accessed. $(MC)^2$ also provides the CPU further opportunities to hide copy latencies through prefetching, accelerating copies even in cases where most data is accessed. In this way, $(MC)^2$ also demonstrates the *hiding* mitigation technique, overlapping the data copy with concurrent CPU computation.

$(MC)^2$'s primary goal is to eliminate copy overheads in the critical path, while ensuring

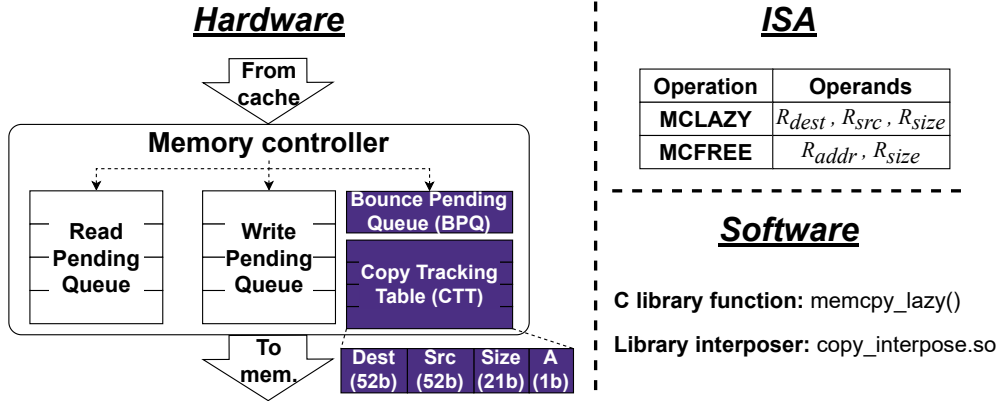


Figure 3.3: Design overview of $(MC)^2$. © 2024 IEEE

data consistency. Figure 3.3 contains all the modifications and features $(MC)^2$ introduces to accomplish these goals. The left side shows the additions we make to the memory controller (shown in purple), while the right side shows the two new ISA instructions we introduce along with the software support that we provide. We refer to a copy that the processor has requested, but not yet serviced, as a *prospective copy*. We use the term *lazy copy* to refer to the process of finally moving the data from source to destination, triggered on an access that requires the copy be performed.

We now cover the hardware changes for $(MC)^2$, look at its software interface, then look at how `memcpy` operations are transparently replaced with their lazy alternative.

3.3.1 Memory controller design

We start by making changes to the memory controller (MC) to support lazy copies.

3.3.1.1 Copy tracking table (CTT)

We add a copy tracking table (CTT) to each MC that tracks prospective copies. The CTT is an SRAM-based module that performs lookups using the physical address of a memory access. These lookups are in parallel with the memory access, avoiding overheads

on the critical path. The CTT ensures that reads to destination buffers are correctly routed (*bounced*) to corresponding source buffers. The source and destination buffers may span multiple memory modules, so we ensure that CTTs across MCs are consistent. This is done by snooping the interconnect for broadcast messages informing the MCs of table modifications.

Table entries. Each entry in the CTT occupies 16 bytes consisting of a 52-bit source physical address, 52-bit destination physical address, 21-bit size, an active bit (as shown in Figure 3.3) and 2 unused bits. Addresses are tracked with 52 bits as this is the upper limit of physical address sizes that most systems support [110, 111]. A 21-bit size allows a single entry in the CTT to track a lazy copy of up to 2MB, the size of a huge page. Many copied buffers do not exceed this size. Further, memory fragmentation causes larger physically contiguous regions to be rare [112], making larger tracking granularities unnecessary.

Table logic. The CTT contains logic to ensure that tracked destination buffers do not overlap. Specifically, if an existing entry contains (part of) a destination buffer that a new operation is inserting, the existing entry is removed (or resized) so that the new and existing entries' destination buffers do not overlap. This corresponds to the case where data is copied to a buffer and then new data is copied to the same buffer. Every destination buffer thus has a unique source buffer.

In addition, when a new entry is inserted into the CTT, we check whether any part of the source buffer of the new entry was a destination buffer in an existing entry. If so, we split the new entry, with the overlapping part using the source buffer of the existing entry. For example, a lazy copy operation is initiated with buffer A being copied to buffer B (copy 1), then buffer B being copied to buffer C (copy 2). The entry in the CTT for copy 2 will show A being copied to C. This avoids chains of copies, simplifying dependence tracking.

The CTT merges multiple lazy memcpy operations to a single entry when it finds copies with contiguous source and destination buffers. This occurs when multiple copies are to logically separate entities within the same buffer, e.g., element-by-element copies of an array.

Required table storage. We allocate 2,048 entries in each CTT to allow it to track a large number of active copies. The CTT access latency is negligible compared to the DRAM

access latency, avoiding overheads in the critical path. Altogether, the CTT uses $2,048 \times 16\text{B}$ or 32KB of SRAM.

Avoiding CTT overflow. If the CPU continuously issues lazy copy operations, the CTT can fill up. To avoid this, copies are performed by the MC asynchronously, with entries freed on copy completion. The tradeoff is that asynchronous copies lead to increased memory bandwidth utilization and reduce the potential of avoiding redundant copies. Conversely, waiting until the CTT is full before copying causes CPU stalls as it waits for free entries.

To strike a balance, $(MC)^2$ starts copying when the CTT becomes 50% full. For this, the MC identifies entries with the smallest size and creates read requests for their source buffers. Once the read is complete, the data is written to the destination buffer and the entry is then removed from the table. For large servers capable of issuing sizeable bursts of copies, the CTT frees multiple entries in parallel, leveraging the increased bandwidth of these servers. This requires only a few bytes for counters to keep track of the number of entries being freed.

3.3.2 Bounce pending queue (BPQ)

Before data in the source buffer can be modified, we must first ensure that data is copied to the destination buffer(s) for correctness. To manage pending writes to source buffers, we extend each MC's existing write pending queue with a *bounce pending queue* (BPQ). The BPQ holds the cacheline(s) containing a corresponding write to a source buffer while the lazy copy to destination is performed. If the number of writes to source buffers exceeds the BPQ size, further writes are stalled by the MC, creating back-pressure on the caches. We find that a small BPQ of 8 entries was sufficient to absorb bursts of source buffer writes.

3.3.3 $(MC)^2$ functionality

We now look at how $(MC)^2$ interacts with the CPU and memory system and how data consistency is preserved while lazily copying.

Requesting a prospective memcopy involves 3 main steps, shown in Figure 3.4. The source buffer is shown in maroon and the destination buffer is shown in purple.

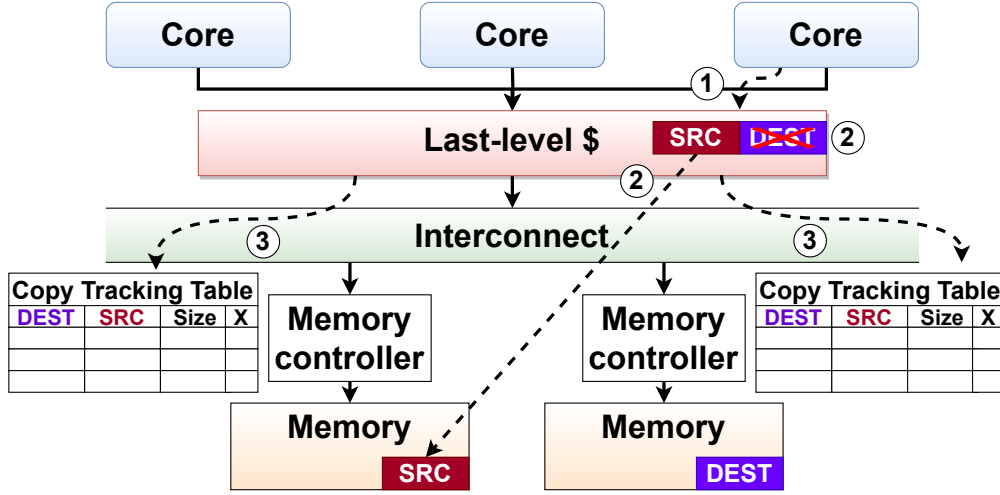


Figure 3.4: Lazy copy operation in $(MC)^2$. © 2024 IEEE

① The CPU issues MCLAZY, creating a packet containing the source buffer address, destination buffer address, and copy size. It sends this packet to the caches.

② Once the packet reaches the caches it triggers writebacks for all the cachelines contained in the source buffer, and invalidates the cachelines contained in the destination buffer. The caches' FIFO write buffer ensures that the writebacks reach the MC before the MCLAZY packet. This guarantees that further MC-observed writes were issued after the lazy copy operation, necessary for memory consistency.

③ The packet is then broadcast across the memory interconnect, and all the MCs insert a new entry containing the details of the lazy copy into the CTT.

3.3.3.1 Memory accesses

The CTT is consulted for every MC-observed memory access. If the memory access is to one of the tracked source or destination buffers, $(MC)^2$ needs to handle it. There are four such types of memory accesses. Namely, reads from destination, reads from source, writes to destination, and writes to source buffers. We discuss each one in detail.

Read from source. As these accesses do not modify data and the source is up-to-date in

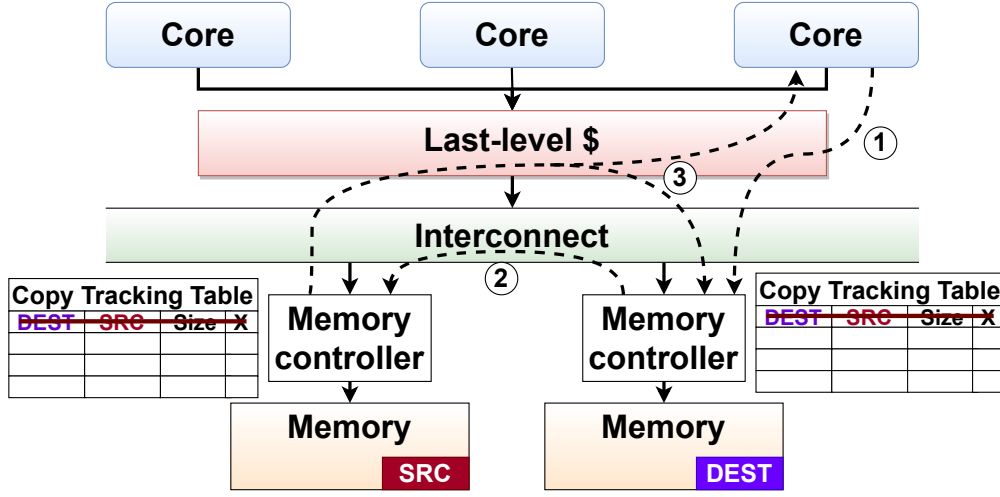


Figure 3.5: Lazy copy on destination buffer read. © 2024 IEEE

memory, they proceed without interference.

Write to destination. If a write to a destination buffer reaches the MC, we no longer need to track it in the CTT, as the memory will now contain fresh data. The MC removes the entry from the CTT (or splits the entry if it spans multiple cachelines). It broadcasts a message on the interconnect for the other MCs to update their CTTs.

Read from destination. As this accesses a prospective copy, the data in the destination buffer in memory is stale. Retrieving the correct data involves 3 steps, shown in Figure 3.5:

① The MC fetches the appropriate source buffer address corresponding to the destination address from the CTT.

② It *bounces* the request, which involves redirecting the request to the MC containing the source, in case the source is within another memory controller. This MC then reads the respective source cacheline from memory, storing the data in the destination response.

③ This packet is sent back to the core as if it was read from the destination. As we now have the up-to-date version of the destination cacheline, a copy of this packet is also sent as a write to the MC containing the destination buffer, eventually removing the corresponding entry from the CTT. This prevents future accesses to the same cacheline from experiencing

further overheads from bouncing.

Write to source. If a write to a source buffer cacheline reaches the MC, we need to execute a lazy copy. To do so, the write is first held in the BPQ and a read to the same source cacheline is generated and sent to memory. Once the source cacheline is obtained from memory, the MC generates packets for each destination cacheline that has a prospective copy with the source. The data read from the source cacheline is copied into these packets. The completed destination cachelines are then written to memory and corresponding CTT entries are removed (or resized). Once complete, the BPQ entries are written to memory.

Unaligned copies. If source and destination copy buffers are not cacheline-aligned with each other, a lazy copy to a destination cacheline may require data from multiple source cachelines. This results in multiple bounces to fetch the entire destination cacheline. For example, if a prospective copy was from physical address 100 (source) to address 512 (destination), we require two bounces to reconstruct the destination; the first access to address [64 - 127] and the second to [128 - 191], which may lie in separate memory modules.

Reducing bandwidth contention. When a destination buffer is read, a copy of the reconstructed cacheline is sent as a write to memory. To avoid contending on memory bandwidth, if the WPQ of the destination MC is more than 75% full, it rejects the write to prioritize the memory bandwidth for accesses from the caches. Otherwise, it writes the destination to memory and removes (or resizes) the corresponding CTT entry. Further reads to this destination cacheline are serviced from memory as normal.

3.3.4 ISA design

We provide two new instructions for programmers to take advantage of lazy memcopy, shown on the right in Figure 3.3.

Lazy copy. MCLAZY enables the lazy memcopy. It takes three register operands. The first register contains the virtual address of the destination buffer, the second register contains the virtual address of the source buffer, while the last register contains the lazy memcopy size. The destination buffer cachelines encompassed by the operation are invalidated while

the source buffer cachelines are written back from the cache.

Alignment requirements. The destination address must be cacheline-aligned and the value contained in R_{size} must be a multiple of the cacheline size. This simplifies the MC logic, as destination cachelines are guaranteed to be lazily copied in their entirety, and avoids partial cacheline invalidations for the destination buffer. We shall see how a software wrapper hides these requirements.

The source and destination must each be contiguous in physical memory. For user-space applications with buffers spanning multiple pages, the instruction must be called separately for each page. The instruction requires at most two address translations, one for the source and one for the destination.

Instruction parallelism. MCLAZY exhibits memory consistency similar to CLFLUSHOPT and CLWB [113]. This means that separate invocations of the instruction proceed in parallel, without serialization required for stores in the x86-TSO memory model [114]. To enforce ordering with future operations, an MFENCE or SFENCE operation must be called.

Freeing. Finally, we provide an MCFREE operation that takes two operands: an address register and a size register. MCFREE sends a hint to the MC that the buffer defined by the address and size registers is no longer useful and can be freed. The MC can remove all entries in the CTT where the destination buffer is contained in the freed buffer. Data in the freed buffer needs to be reinitialized before reuse, i.e., a read operation to a freed buffer following an MCFREE call is undefined. This instruction can be called within functions like `munmap` where the buffer is guaranteed to no longer be used.

3.3.5 Software design

C/C++ function. To remove constraints on the programmer, we provide a C/C++ library function `memcpy_lazy` that has the same semantics as a standard `memcpy` call. Internally, this function calls MCLAZY for each page in the buffers and ensures that the destination is cacheline-aligned.

The pseudocode for this function is shown in Figure 3.6. Here, `CL_SIZE` refers to the

```

1  def memcpy_lazy(dest, src, size):
2      # Cacheline align dest
3      left_fringe = ALIGN_REM(dest, CL_SIZE)
4      memcpy(dest, src, left_fringe)
5      dest += left_fringe
6      src += left_fringe
7      size -= left_fringe
8      while size > 0:
9          # Calculate remaining size in page
10         src_off = ALIGN_REM(src, PAGE_SIZE)
11         dest_off = ALIGN_REM(dest, PAGE_SIZE)
12         # Pick minimum size left as lazy copy size
13         copy_size = min(min(src_off, dest_off), size)
14         if copy_size < CL_SIZE:
15             memcpy(dest, src, copy_size)
16         else:
17             # Make copy_size a multiple of CL_SIZE
18             copy_size &= ~(CL_SIZE - 1)
19             MCLAZY(dest, src, copy_size)
20         dest += copy_size
21         src += copy_size
22         size -= copy_size
23     mfence()

```

Figure 3.6: Lazy memcpy function pseudocode.

cacheline size, while the macro `ALIGN_REM` returns the number of bytes required to align a given address to a given alignment size. First, we identify the number of bytes needed to be added to the `dest` address to cacheline-align it (line 3). We then copy these bytes from the source to destination (line 4). The destination is now cacheline aligned. We then identify the number of bytes remaining in the source and destination pages (lines 10 - 11) and pick the smaller one (line 13). This ensures that the copy is being performed on a single page for both the source and destination. If this is smaller than the cacheline size, we manually memcpy it (line 15). Otherwise, we make the copy size a multiple of the cacheline size and call `MCLAZY` for it (lines 18 - 19). We then repeat the process until the entire copy is complete. We finally issue an `mfence` call to order the prospective copies with future memory accesses (line 23). The read and write operations of a standard `memcpy` are replaced by a `MCLAZY` call per copied buffer page, enabling our lazy memcpy goal.

Memcpy interposition. We provide an additional interposer library (`copy_interpose.so`) that converts memcpy calls to lazy memcpy calls for legacy applications. This removes the

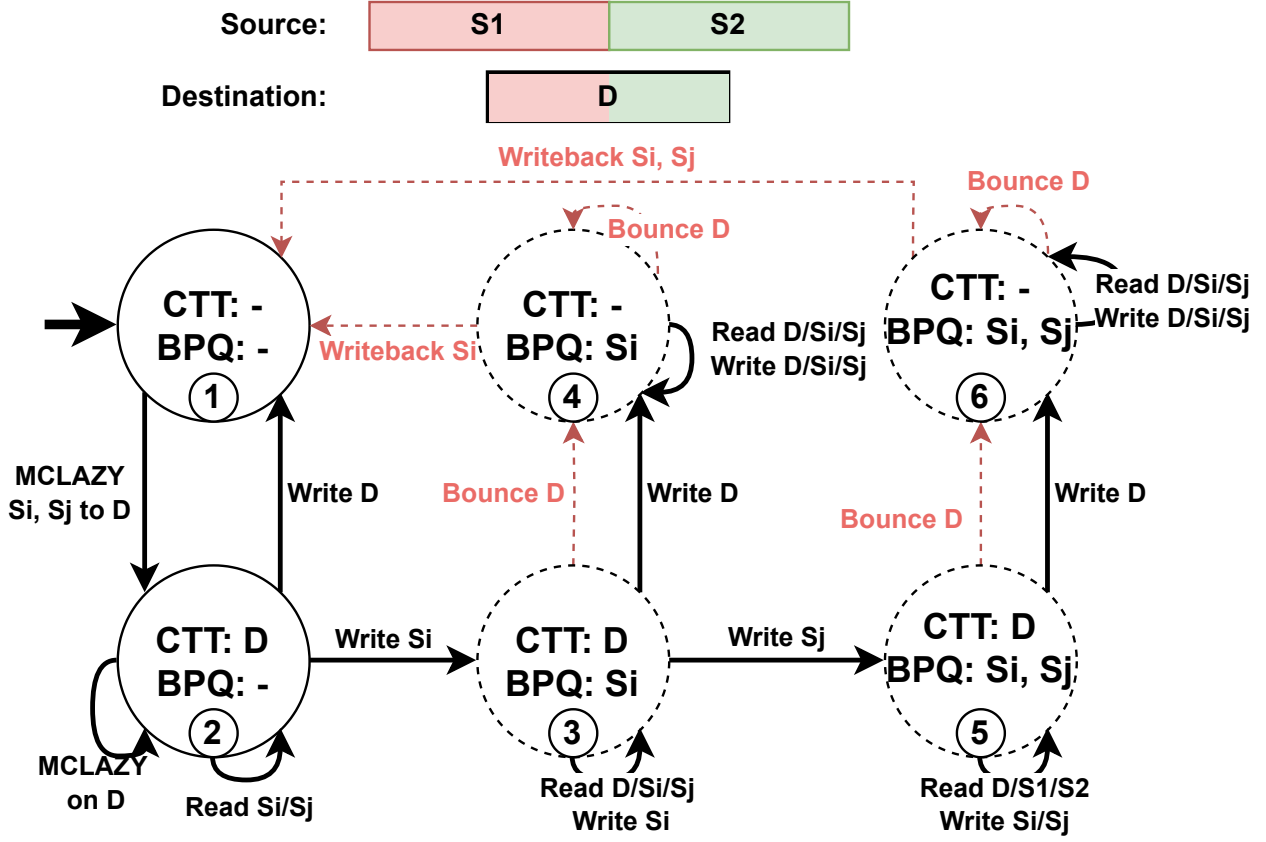


Figure 3.7: $(MC)^2$ simplified state transition diagram.

need to rewrite code to take advantage of $(MC)^2$.

3.3.6 Data protection and correctness

We now look at $(MC)^2$'s data protection and correctness guarantees. For concreteness, we assume the x86 CPU architecture and x86-TSO memory model [114], although $(MC)^2$ may be applied to other models with minor modifications.

Protection. As $(MC)^2$ deals with only physical addresses, regular address translation occurs within the CPU where page protection bits are checked. If a process tries to copy data it does not have access to, the MMU raises a page fault. On unmapping pages, the operating systems zeroes out the newly freed pages before providing them to new processes.

This reaches the MC as writes, removing entries from the CTT—avoiding data leaks from occurring between processes. This ensures that $(MC)^2$ does not expose data protection risks.

Memory consistency. $(MC)^2$ must ensure that reads and writes to source and destination buffers preserve memory consistency. In $(MC)^2$, each destination cacheline behaves independently. This destination cacheline is a prospective copy of a single source cacheline if they are cacheline-aligned. In cases of misalignment, the destination is split across two source cachelines. To examine the interaction of memory accesses with $(MC)^2$, we use a state transition diagram (Figure 3.7). States ① - ④ cover the transitions for when source and destination are cacheline-aligned. When misaligned, we have additional states ⑤ - ⑥.

Assume that S1, S2 and D are each cachelines, with S1 and S2 being contiguous physical addresses. Part of S1 and S2 are prospective copies to the destination cacheline D, meaning part of D’s data lies in S1 and part in S2, as seen in the top of Figure 3.7. Note that S1, S2, and D may all lie in different memory modules. The different states in the transition diagram show the different possible states of the BPQ and CTT. State transitions are caused by memory accesses from the CPU [in black] or $(MC)^2$ [in red]. To understand how $(MC)^2$ guarantees memory consistency, we shall now go through each state of the diagram and analyze all the interactions and transitions to see how the system reacts in each case.

① We start with an empty CTT and BPQ. In this state, $(MC)^2$ has no impact. On receiving a prospective copy, we transition to state ②.

② In this state, the CTT contains an entry noting that S1 and S2 have been lazily copied to D. Writing to D removes the entry from the CTT and transitions us back to state ①, while reading S1 or S2 has no impact. Performing another prospective copy with destination D retains us in the same state, with the CTT entry being modified to contain the new source(s). On writing to either S1 or S2 (denoted by Si), we move to state ③, where the write is kept in the BPQ.

③ This is a transitional state, where a bounce packet is generated which reads S1 and S2 from memory (and not the BPQ) and then writes to cacheline D. On completion of the write, we move to state ④, denoted by the “Bounce D” transition. A similar transition

occurs when the CPU directly writes to D. Reads and writes to S_i issued by the CPU are merged and serviced directly from the BPQ. On a write to S_j ($j \neq i$) we move to state ⑤.

④ This is also a transitional state. D is removed from the CTT as its data has been retrieved and written to memory, i.e., the lazy memcpy is complete. Reads and writes to D are serviced normally by the memory as the CTT does not contain an entry for it. The BPQ writes S_i to memory to move back to stable state ①. If S_i is a common source for multiple prospective copy destinations, e.g., D1, D2, D3..., the BPQ must wait until all entries with S_i as source are removed from the CTT before writing S_i to memory.

⑤ This is also a transitional state, where both S_1 and S_2 are kept in the BPQ (denoted by S_i and S_j). Similar to state ③, all reads and writes to S_1 and S_2 from the CPU are serviced by the BPQ, while the bounce packet for D reads directly from memory. Writing to D or completion of the bounce operation moves us to state ⑥.

⑥ Similar to state ④, D is removed from the CTT. S_1 and S_2 are written back to memory to move back to state ①. It may occur that S_1 and S_2 lie in different memory modules operated by different MCs, with both independently generating identical bounce requests for D. One may complete, while the other is still pending. For this, bounce requests for D are dropped on reaching this state.

In all the transitional states where the BPQ contains either S_1 or S_2 (③ - ⑥), prospective copies involving S_1 or S_2 are stalled until S_1 and S_2 have been written back to memory.

3.4 Evaluation methodology

We simulate the performance of $(MC)^2$ by extending GEM5-v22.1 [115], a cycle-level CPU simulator. Table 3.1 gives the configuration details of our simulation. Our system resembles a scaled-down server node.

We also used CACTI 7.0 [116], a tool that calculates cache specifications based on provided parameters, to obtain the CTT access latency and area. For our configuration, with a 22 nm transistor size, we find that the CTT area is 0.14 mm^2 which is negligible compared to I/O die areas of 100 mm^2 [117]. The bank leakage power is 33.8 mW. The CTT latency is

Table 3.1: Simulated configuration. © 2024 IEEE

Hardware			
CPU s	8	Clock speed	4 GHz
Private L1 cache	64 KB/CPU, Stride prefetcher	Shared L2 cache	2 MB, Stride prefetcher
DRAM size	3 GB	DRAM channels	2
DRAM config.	DDR4	BPQ size	8 entries
CTT entries	2,048	CTT latency	0.79 ns
Software			
OS kernel	Linux 5.7.0	Distribution	Ubuntu 20.04

0.79 ns, significantly lower than typical DRAM access latencies (15 - 90 ns [76]). The CTT latency is incurred when the destination of a prospective copy is read. The entry in the CTT is looked up and the ongoing access is preempted, incurring the CTT latency. The packet is then bounced towards the source.

Throughout our evaluation we make use of our C wrapper function `memcpy_lazy`. To accurately model the performance of cacheline writebacks required by `MCLAZY`, this function calls the `CLWB` instruction [113] for each cacheline that needs to be written back. It also cacheline-aligns the provided destination, calling `memcpy` for unaligned fringes.

Baselines. We compare our approach to a baseline `memcpy` operation. We also compare against `zIO` [38], a state-of-the-art approach for OS-assisted zero-copy IO. `zIO` elides `memcpy` operations and tracks the copies in a skiplist. The page table entries are marked as copy-on-access using `userfaultfd` [118]. On a page fault, `zIO` allocates physical memory for the destination buffer and performs the copy. We modify `zIO` to perform elision on all `memcpy` calls instead of just IO-based copies.

3.5 Evaluation

We evaluate $(MC)^2$ and tease out the performance implications of the lazy `memcpy` technique through a set of microbenchmarks. We then look at the full-system performance of

$(MC)^2$ through benchmarks and applications consisting of Google’s Protobuf, MongoDB, and Cicada. We also analyze the impact of $(MC)^2$ on Linux huge page faults. We conclude by examining $(MC)^2$ ’s sensitivity to different configuration parameters.

Our evaluation answers the following questions:

1. How much lower is $(MC)^2$ ’s copy overhead in the critical path? (§3.5.1)
2. What is the impact of *lazily* copying data upon access? (§3.5.2)
3. What benefit does $(MC)^2$ provide to applications? (§3.5.3)

3.5.1 Copy latency

We examine copy latency of $(MC)^2$ by performing `memcpy` operations on various sizes of data regions prefaulted into memory.

Figure 3.8 shows the latency for zIO `memcpy` elision and $(MC)^2$ lazy `memcpy` compared to a baseline of native `memcpy` (lower is better). $(MC)^2$ enables cacheline-sized lazy copies, with speedups for copies 1KB and larger. It has a significantly lower overhead than zIO for smaller copies, and is 55% to 11× faster than `memcpy` for copies of a kilobyte and larger. As zIO relies on page table copy-on-access for elision, it requires copy sizes of at least a page to be able to perform elision. The overhead of unmapping pages and issuing TLB shootdowns ends up degrading performance for smaller copy sizes, with zIO performing worse than native `memcpy` for 16KB copies. zIO’s cost becomes justified for copy sizes of 64KB or larger, with a speedup of 23× over `memcpy` at 4MB. Despite this, we shall see that when the destination buffer is accessed, zIO suffers mis-speculation penalties, degrading the overall performance.

We also analyze the performance of `memcpy` when the source buffer has already been touched before the operation, leading to data being cached (Touched `memcpy`). This is the best possible performance, as the buffer is already in the cache if it fits. We see that this outperforms $(MC)^2$ for smaller sizes, however, for 16KB and above $(MC)^2$ is able to provide

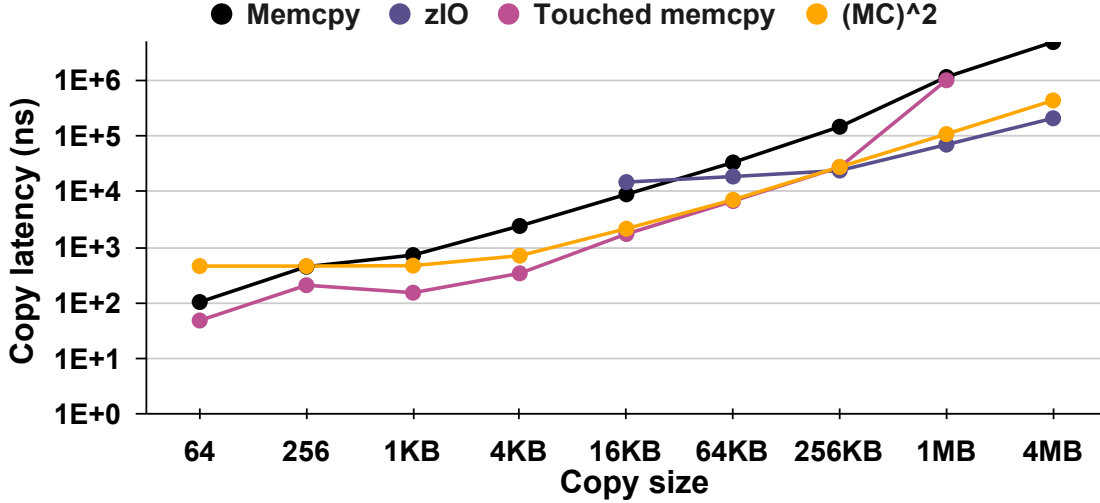


Figure 3.8: Copy latency for native memcpy, zIO and $(MC)^2$. © 2024 IEEE

a similar memcpy latency. This shows that $(MC)^2$ can be used to provide copy latencies similar to cached copies, regardless of whether the data is present in the cache.

$(MC)^2$ lazily copies data when either the destination buffer is read or the source is written. This can lead to overheads in the critical path when the destination is read. We analyzed this alongside optimizations and hardware features that reduce this impact.

3.5.2 Data access latency

In Figure 3.9 we measure the time taken to copy a 4MB buffer, followed by reading the destination sequentially. This imitates a streaming access pattern frequently found in workloads like serialization. Recall that $(MC)^2$ causes reads to the destination to bounce. We misaligned the data so that this bouncing has the highest penalty of two bounces.

Initially, zIO is 70% faster than $(MC)^2$ when no data is accessed. However, as the buffer gets accessed, this flips, as zIO is forced to handle page faults due to its copy-on-access mechanism for copy elision. When at least half of the dataset is accessed, zIO’s overhead causes it to perform worse than memcpy. Conversely, $(MC)^2$ always outperforms native memcpy, with a worst case of 80% of the runtime of memcpy.

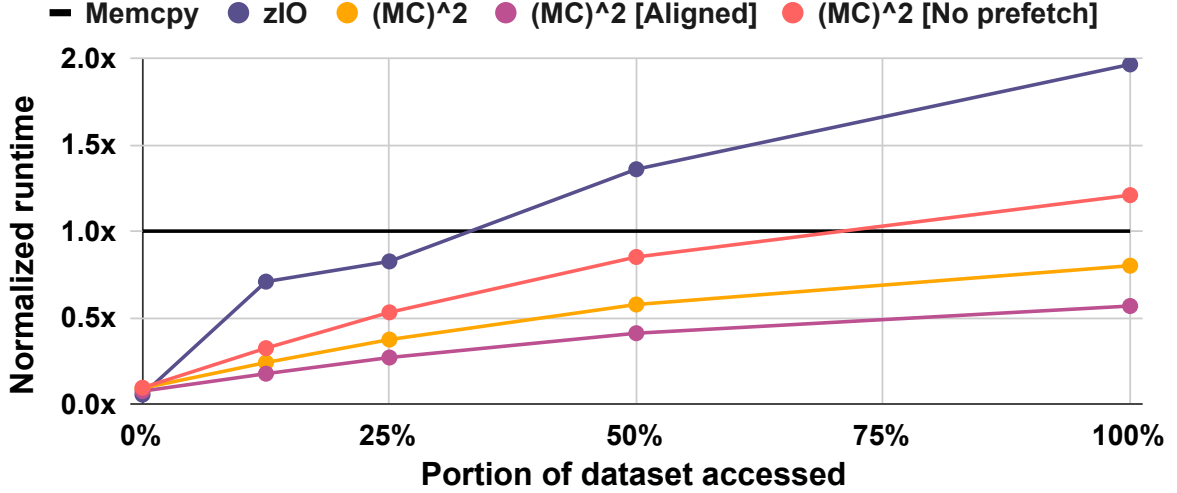


Figure 3.9: Normalized runtime for sequential accesses of destination buffer. © 2024 IEEE

Hardware prefetchers are able to predict the sequential access pattern, generating accesses to the destination before the CPU requests them. This hides the latency of bouncing, which enables the speedup of $(MC)^2$ relative to memcopy. When the hardware prefetcher is turned off, we see $(MC)^2$ performs up to 21% worse than memcopy. This is how $(MC)^2$ also demonstrates the *hiding* mitigation for data movement.

For random accesses, we perform a pointer-chasing experiment, whose results are shown in Figure 3.10. zIO suffers frequent page faults, peaking with $2.1\times$ runtime at 12.5% of buffer access. When more of the previously faulted pages are re-accessed, this overhead starts to reduce to $1.3\times$ that of memcopy.

On the other hand, $(MC)^2$ has only 92% of the runtime of memcopy, as accessing a destination cacheline for the first time causes the copy to be performed, but on every further access to the same cacheline, the access proceeds as normal. If the buffers were cacheline-aligned, then $(MC)^2$ performs at a worst case runtime of 88% of memcopy.

3.5.3 Application workload evaluation

We now look at how $(MC)^2$ performs on real application workloads.

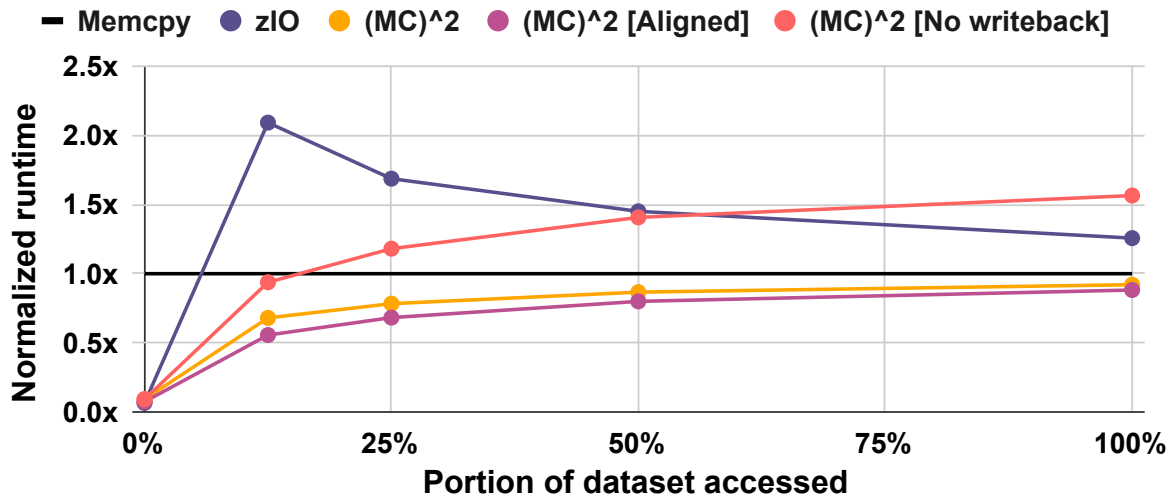


Figure 3.10: Normalized runtime for random accesses of destination buffer. © 2024 IEEE

Google Protobuf. To evaluate the effectiveness of $(MC)^2$ on a practical workload, we ran the Protobuf workload provided in the Google Fleetbench [86] benchmark suite. Fleetbench consists of workloads dedicated to common “hot” library functions using traces obtained from production servers. Google’s Protobuf library [81], which provides a language-agnostic framework for data serialization, is a major workload in Fleetbench. The Protobuf workload calls different Protobuf functions with message sizes obtained from server traces. We accel-

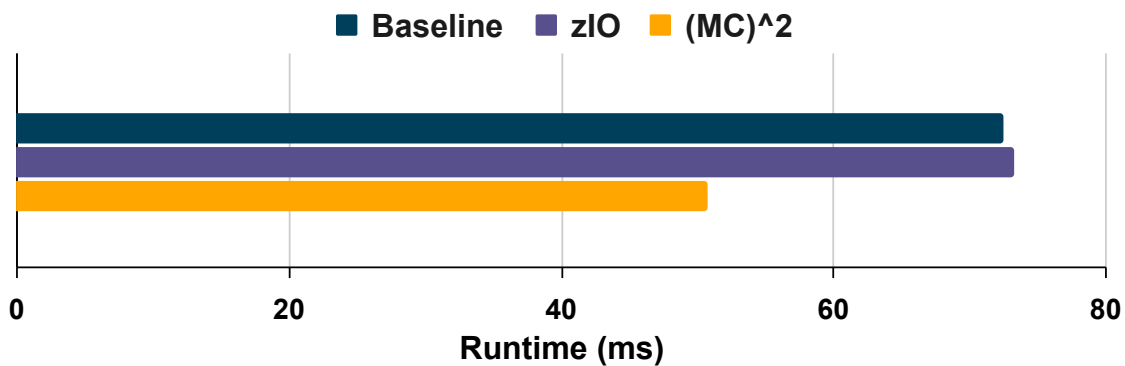


Figure 3.11: Runtime of Protobuf. © 2024 IEEE

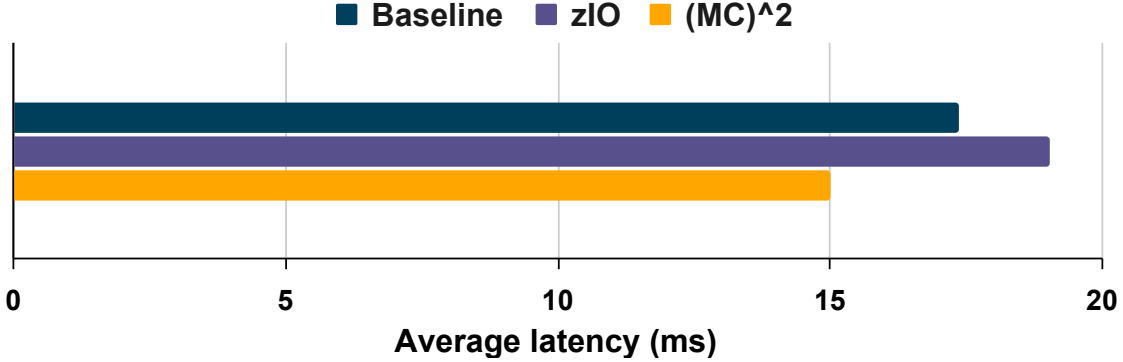
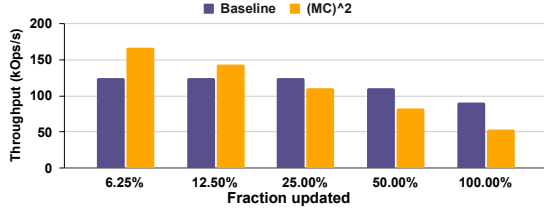


Figure 3.12: MongoDB average insertion latency. © 2024 IEEE

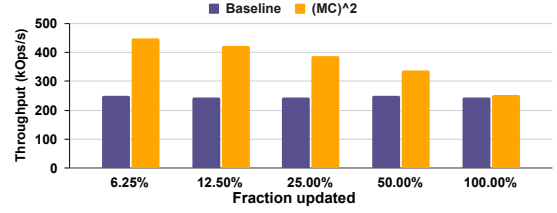
erate this workload with $(MC)^2$ using the library interposer to redirect `memcpy` calls 1KB and larger to our `lazy_memcpy` function.

In Figure 3.11 we see the runtime obtained for the baseline Protobuf workload compared to the runtimes of the workload with zIO and $(MC)^2$. We find that all `memcpy` operations were below page size preventing zIO from performing any elision. $(MC)^2$ provides 43% speedup over `memcpy` by eliding the copies during serialization that do not end up accessed. **MongoDB.** We examine $(MC)^2$'s effectiveness in avoiding redundant copies in IO buffers by running MongoDB [87] with $(MC)^2$ and zIO's MongoDB copy elision interposer [119]. We replicate the experiment performed by Stamler et. al. [38], where a client runs the YCSB [120] load phase with 100KB fields and 10 fields per insertion. The load phase performs 100% inserts in a uniform random distribution. We scale down the number of insertions to 50 to make the simulation time feasible. We run this workload 3 times and report the average insertion latency in Figure 3.12. $(MC)^2$ speeds up insertions by 15.5%, while zIO slows down insertions by 9.7% due to frequent accesses to copied data.

For large copy sizes, zIO is supposed to provide better copy latencies than $(MC)^2$ (§3.5.1). Despite this, $(MC)^2$ provides better performance than zIO as it does not have the page fault penalties that zIO experiences when prospective copies are accessed. MongoDB copies data into an in-memory B-tree used for indexing and into a log during transaction commit. During



(a) 1 thread.



(b) 8 threads.

Figure 3.13: Read-modify-write MVCC database throughput. © 2024 IEEE

this process, it accesses copied data. For zIO, this triggers a page fault, forcing it to perform a copy on the accessed page, which $(MC)^2$ avoids.

Multi-version concurrency control. Multi-version concurrency control (MVCC) [10] is a popular database transactional consistency technique. With MVCC, write transactions that modify data first create a local copy of the tuples being modified. Only the local copies are modified, creating a new version of the data upon commit. These new versions are hooked back into the main database, discarding the old data. This allows concurrent read transactions to read consistent data from the main database without requiring locks.

Transactions often only update a small portion of a tuple, incurring unnecessary copy overhead. For example, TPC-C, an industry-standard OLTP benchmark, has frequent operations that update a single tuple attribute (e.g., decrementing the quantity of stock of an item [121]) which updates only 1–2% of the tuple. Tracking at a sub-tuple granularity drastically increases the complexity of version management and is thus avoided by databases [10].

$(MC)^2$ allows MVCC databases to utilize tuple-wise copying, while paying the copy penalty only for the portions updated. To demonstrate this, we enhance the Cicada MVCC database [88] with $(MC)^2$. We perform repeated operations on a table with 8KB-sized rows, modifying different fractions of the tuples and measuring throughput. The operations are split in a 50:50 ratio between reads and updates, typical of write-intensive database workloads [120]. We ran this experiment with one and eight threads performing transactions to analyze performance when latency-bound and bandwidth-bound respectively.

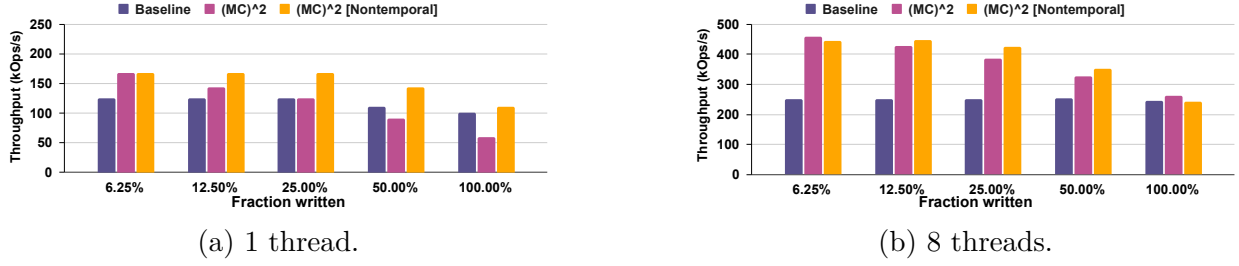


Figure 3.14: Write-only MVCC database throughput. © 2024 IEEE

Figure 3.13 shows the transaction throughput of baseline Cicada compared to $(MC)^2$ when the updates are read-modify-write (RMW) operations. The baseline reads data from memory during `memcpy` then performs the RMW locally in the cache. $(MC)^2$ avoids the memory read during `memcpy`, and only reads from memory the fraction of data being updated during the RMW operation. For updates that modify less than 25% of the tuple, $(MC)^2$ provides up to 78% higher throughput. For higher fraction of updates with one thread, the memory read penalty of $(MC)^2$ outpaces the copy speedup. With eight threads the transactions are memory bandwidth-bound, and as $(MC)^2$ reduces memory accesses it consistently provides performance improvement for update fractions less than 100%.

Figure 3.14 shows the transaction throughput when the update operations are write-only. The throughput mimics that of RMW because cache write misses issue read-for-ownership (RFO) [113] that reads data from memory before writing to the cache, incurring $(MC)^2$'s read penalty. If the store operations are replaced by non-temporal stores [113] (uncached stores that avoid RFO), we see that $(MC)^2$ provides a higher throughput with one thread than the baseline for all write fractions. Similar to the previous case, for 8 threads the application becomes memory bandwidth bound, leading to performance improvements until the entire tuple is modified. We were unable to compare to zIO as Cicada allocates memory using `MAP_SHARED`, which zIO does not support.

Concurrent snapshots with huge pages. In-memory databases make use of virtual memory snapshotting [89] to take concurrent database snapshots. They leverage the `fork`

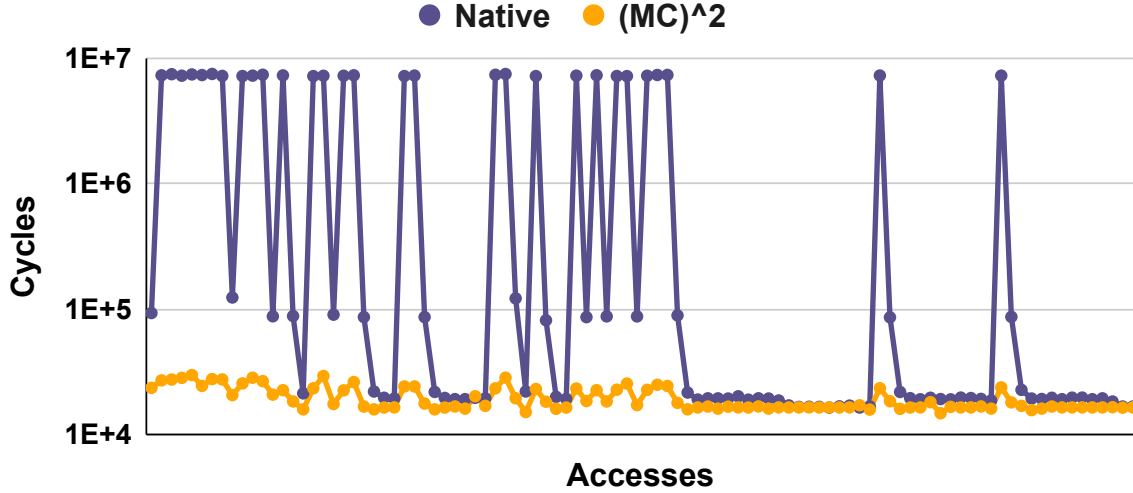


Figure 3.15: Write latencies with huge page copy-on-write. © 2024 IEEE

system call to create a virtual memory snapshot in a child process. To minimize overhead, `fork` does not copy memory to create the snapshot. Instead, the parent and child map the same memory as copy-on-write, copying memory pages lazily on writes [39].

With their large datasets, in-memory databases would like to use huge page mappings to minimize TLB misses. However, despite the use of huge pages reducing the direct overhead of `fork` by over an order of magnitude due to smaller copied page tables [122], huge page copy-on-write faults require larger 2MB copies, causing significant latency spikes. For this reason, many in-memory databases advise against huge pages [91, 123].

To demonstrate how $(MC)^2$ can mitigate this issue, we modify the Linux kernel’s huge page copy function to use MCLAZY instead of copying the huge page immediately. We run a program that initializes a 64MB memory region using huge pages, calls `fork`, and then updates random 8-byte elements in the 64MB region. We measure the latency of each update using the RDTSC [113] instruction.

Figure 3.15 shows the latencies of the first 100 accesses, where Native is the unmodified Linux kernel, and $(MC)^2$ uses our modified kernel. The native kernel experiences latency spikes up to $455\times$ during page faults. $(MC)^2$ experiences spikes at most $2\times$ with worst-case

latencies $250\times$ lower than native. This demonstrates how $(MC)^2$ reduces latency spikes caused by huge page copy-on-write faults by over two orders of magnitude, while retaining the benefits of huge page mappings.

3.6 Performance tradeoff discussion

While $(MC)^2$ provides the same semantics as the standard memcopy and aims to accelerate it, there are performance tradeoffs. We discuss them in this section.

Cached source buffers may harm performance. When the source buffer is already present in the cache, the overhead of copying is low as the CPU fetches data from the cache instead of memory. Replacing these copies with $(MC)^2$ could harm performance. Despite this, we saw in subsection 3.5.1 that the latency of $(MC)^2$ is not significantly worse than a cached copy in most cases.

Reduced cache pollution. Cache pollution is a common problem associated with buffer copies [124, 55], where the destination of the copy is not immediately accessed after the copy. As $(MC)^2$ explicitly invalidates destination buffers from the cache, it avoids this problem.

Copy-and-access may harm performance. If a destination buffer is immediately accessed after a prospective copy, $(MC)^2$ could reduce performance, as the destination buffer’s cache lines have been invalidated. We saw in section 3.5 that prefetching eliminates this performance impact in many cases.

Memory footprint. Prior work, such as zIO [38], performed copy elision by unmapping destination pages and marking them copy-on-access, reducing the memory footprint for copies with unaccessed destination buffers. $(MC)^2$ does not reduce memory footprint, as both source and destination buffers must be physically allocated by the OS before lazy copying. This allows $(MC)^2$ to remain transparent to the OS and support lazy copies at sub-page granularity, avoiding hardware interrupts and page faults on access to destination buffers. For an application like Protobuf, we found that zIO provided no memory footprint reduction due to sub-page sized copies. For MongoDB, we found only a modest 6 – 8% reduction in memory, caused by copied data frequently being accessed.

3.7 Conclusion

$(MC)^2$ embodies the *avoiding* data movement mitigation technique by supporting a lazy copy operation that avoids overheads for copied data that is not accessed, providing benefits across a variety of applications. We place $(MC)^2$ at the memory controller allowing it visibility into all memory accesses that miss the caches—typically the largest source of copy overhead. We demonstrate improvements across a diverse set of workloads such as Protobuf, MongoDB insertions, Cicada writes, and huge page copy-on-write faults. In the next chapter we shall discuss POD-Attention, which addresses the GPU memory bandwidth bottleneck during LLM inference through a different method: hiding memory access overheads behind concurrent computation.

Chapter 4

POD-ATTENTION: UNLOCKING FULL PREFILL-DECODE OVERLAP FOR FASTER LLM INFERENCE

In this chapter we discuss POD-Attention, which demonstrates the *hiding* technique by overlapping compute-bound and memory-bound phases of LLM inference. We apply SM-aware CTA scheduling, which guarantees that the two operations are co-located within each GPU streaming multiprocessor (SM). This ensures that both the compute and memory units present within each SM get utilized concurrently by the two complementary phases.

The infrastructure for serving large language models (LLMs) is ever-expanding to meet growing demands [125, 126]. Large-scale service providers depend on expensive high-end GPUs to meet peak demand or latency targets [127]. Optimizing LLM serving systems has become crucial [128, 129, 130, 131, 58, 132, 133], with the overall efficiency of a deployment being dependent on how well GPU resources are utilized.

From a resource utilization perspective, LLM inference is challenging because different phases require different resources at different times [19, 130, 58, 134]. LLM inference involves two distinct phases: (1) prefill, where the LLM takes the user input and processes it, and (2) decode, where the model generates outputs token-by-token. A major overhead of inference is *attention* [135, 13, 136], which is used to weigh the relative importance of the tokens. Prefill attention operates on large input matrices and is compute-bound. Decode generates individual tokens, converted to vectors or small matrices, with minimal data reuse [137] and is thus memory bandwidth-bound. Serving LLMs efficiently requires both high compute and high memory bandwidth. An ideal system would strive to maximize the utilization of both compute and memory. However, doing so is non-trivial because for a given request, the prefill and decode phases occur at different times.

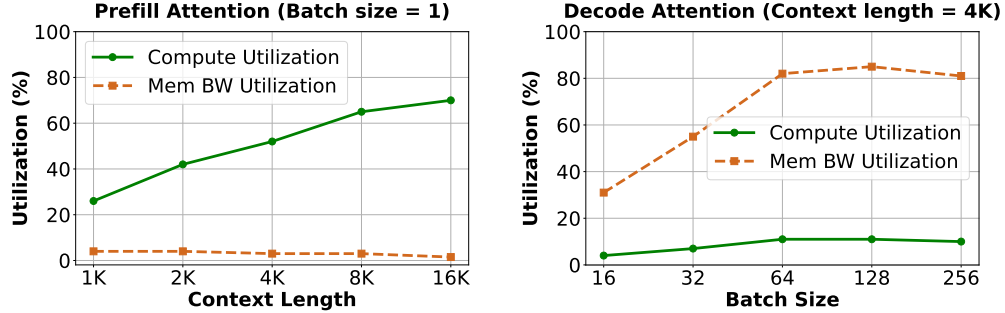


Figure 4.1: State-of-the-art FlashAttention kernels utilize either compute or memory (Model: Llama-3-8B on 2 A100 GPUs).

Modern LLM inference systems make use of hybrid batching [130], which combine prefills and decodes of different requests into mixed batches to improve the overall throughput by reusing the same model weights for both. Hybrid batching also helps reduce tail latency: to limit the runtime of each iteration, the scheduler divides long input prompts (prefill inputs) into multiple smaller chunks, then combines ongoing decodes with a new prefill chunk every iteration [130, 138]. As such, use of hybrid batching is common in various LLM serving systems today [129, 58, 130, 128, 138].

While prior work has focused on optimizing the linear operations [130, 128, 138], they do not optimize the attention computation of a hybrid batch. This is reasonable for a system that primarily deals with small context lengths since linear operations dominate run time in this setting [128, 58]. In contrast, as the context length increases, attention computation becomes the primary performance bottleneck.

Recent work has optimized attention computation [135, 25, 137, 13], but current solutions address prefill and decode operations separately—maximizing compute utilization for prefills and bandwidth utilization for decodes, as shown in Figure 4.1. In this chapter, we show that such an approach is suboptimal as it leaves critical GPU resources underutilized during different parts of computation. For example, Figure 4.1 illustrates that memory bandwidth utilization of the prefill attention kernel is often below 5%, while compute utilization of the

decode attention kernel is under 10%. The effect of using independently optimized kernels is particularly noticeable with hybrid batching because prefill and decode kernels execute immediately one after the other.

To improve the efficiency of hybrid batching, we present POD-Attention—the first GPU kernel that efficiently batches the computation of prefill and decode attention. Existing techniques do not provide adequate performance in fusing attention computation due to various limitations such as straggler threads, synchronization barriers and lack of guaranteed co-location of different Cooperative Thread Arrays (CTAs) on GPU Streaming Multiprocessors (SMs). Instead, our kernel maximizes both compute and memory bandwidth utilization by carefully splitting the GPU’s resources at a fine granularity to co-locate batches of prefills and decodes within the same SM. This allows compute-heavy prefill operations to *hide* the latency of bandwidth-hungry decode, improving overall throughput. Unfortunately, guaranteeing SM co-location is not a trivial task. Scheduling of kernels on SMs is handled by the GPU driver and hardware, not by the programmer.

To achieve this, we introduce a software-based CTA scheduling scheme within the GPU to appropriately co-locate the prefill and decode CTAs. In addition, we tune the prefill and decode functions, reduce register pressure, and appropriately select tile sizes for both operations to minimize interference caused by this new co-location. POD-Attention significantly accelerates attention computation by utilizing both compute and memory resources as per the requirement of a given batch of requests.

Evaluation on A100 GPUs indicate that this kernel provides up to 59% latency improvement over state-of-the-art attention kernels like FlashAttention-2 [139], FlashDecoding [25], and FlashInfer [26]. We’ve integrated this kernel into the vLLM serving stack, seeing improvements of up to 27% in end-to-end LLM inference throughput over vLLM for the LLaMa-3-8B [140] and Yi-6B [141] models.

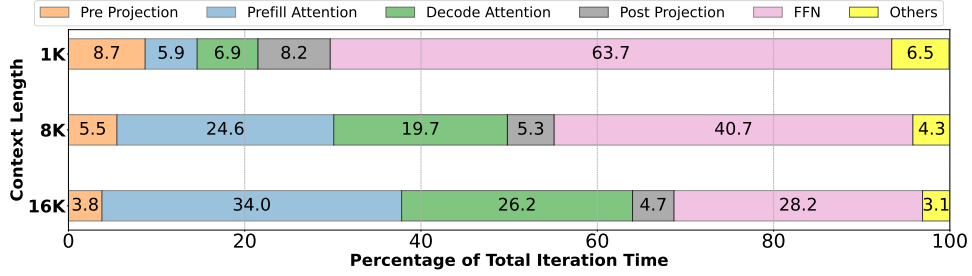


Figure 4.2: Contribution of different operations in iteration runtime with hybrid batching (model: Llama-3-8B, batch size: 60, chunk size: 1K).

4.1 Motivation

Attention computation for prefill and decode is typically serialized in hybrid-batching systems, as requests operate on discrete data items (key-value entries). Separate prefill-optimized and decode-optimized GPU kernels are used which provide better performance than running both using either kernel [13, 26].

Despite each individual attention kernel saturating specific GPU resources, the GPU’s compute and memory bandwidth are still significantly underutilized, shown in Figure 4.1. As the context length (size of input prompt and output) of LLMs increases, the attention operations start dominating the latency of inference, by up to 60% as shown in Figure 4.2. Prefills only use $\sim 5\%$ of available memory bandwidth, while decodes only consume $\sim 10\%$ of compute. Note that prefill and decode attention are computed immediately one after the other in hybrid batches. Therefore, *when independently optimized attention kernels are used, GPU execution goes through periods of high demand of a resource followed by low utilization of the same resource*. For example, the prefill kernel requires high compute but compute is (mostly) idle when the decode kernel executes. We ask the question: is it possible to saturate both resources concurrently to improve throughput?

4.2 Background and related work

We shall briefly cover the GPU’s execution hierarchy and basics of LLM inference, necessary to appreciate this work, then cover related work.

4.2.1 GPU execution model

The GPU’s hardware is arranged in a hierarchy that supports execution at a scale of hundreds of thousands of parallel threads, depicted in Figure 4.3 [142]. The main processor unit of a GPU is a *Streaming Multiprocessor (SM)*, with modern GPUs containing around a hundred SMs. Each SM has an L1 cache and *shared memory* along with tensor cores for accelerated general matrix multiplication (GEMM) and execution units for integer/floating point operations. The shared memory is a user-addressable partition of the L1 cache. The GPU memory is accessed by SMs through the shared L2 cache.

GPU programming languages expose a hierarchy of threads that mimic the hardware hierarchy. The smallest unit of execution is a thread, while a group of 32 threads make up a *warp*, which typically execute in lockstep. To maximize throughput, GPU programmers ensure that threads within a warp execute the same code path. A *Cooperative Thread Array (CTA)* [143] is a group of warps that share the L1 cache and shared memory. All warps in a CTA are guaranteed to execute within a single SM.

Users launch GPU *kernels*, or GPU-executed functions, specifying the number of threads in the CTA, the number of CTAs in the kernel, as well as the required shared memory per CTA. This launch is then queued in a *stream*; operations within a stream are serialized but different streams can execute in parallel in any order. The *CTA scheduler* selects CTAs from streams and assigns them to SMs when sufficient execution resources (e.g., threads, shared memory and registers) are available within the SM.

Central to the GPU’s massive throughput is the fast, cycle-level *warp scheduler* baked into the hardware. Every clock cycle, the warp scheduler dispatches eligible warps for execution. A warp is eligible if its threads aren’t stalled (e.g., waiting for memory access) and the

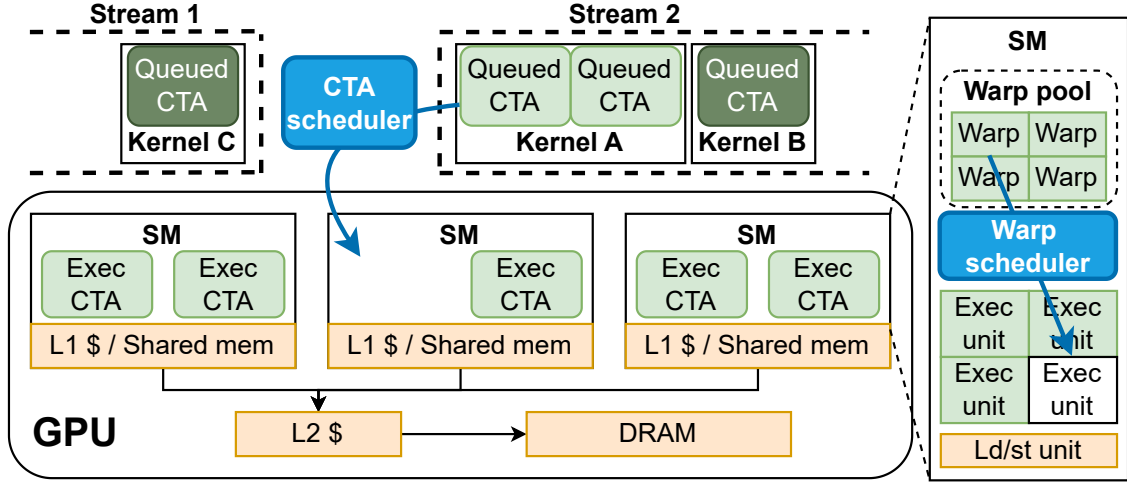


Figure 4.3: GPU execution model.

resource required for its current instruction (e.g. tensor core) is idle. This allows each SM to context switch at every clock cycle, effectively utilizing all its execution resources.

4.2.2 LLM inference

LLMs process text converted to the form of tensors by model tokenizers. The fundamental unit of text (roughly a word) converted to tensor form is referred to as a token. The two major operations of LLM inference are feedforward networks, consisting of matrix-matrix multiplications between tokens and the model weights, as well as attention, consisting of sequence-level operations between each token and every other token in the request.

Inference undergoes two distinct phases, prefill and decode. Prefill involves reading the user's prompt to produce the first output token, whose latency is referred to as the time-to-first-token (TTFT). This newly generated token is then fed back into the model, starting the decode phase, which continuously generates output tokens one at a time in an autoregressive fashion. The latency to generate each output token is called the time-between-tokens (TBT).

The prefill phase operates on a large number of tokens at once, and so is highly parallelizable and compute bound. Decode operates on a token at a time, and so is more limited

in parallelism and memory bound. Decode relies on batch processing to make adequate use of the parallelism provided by GPUs.

Hybrid batching [144] is used to balance compute and memory bandwidth usage. A portion of a request’s prefill is combined with a batch of other requests’ decodes. This reuses the model weights for the batch of requests for feedforward networks, improving performance. Unfortunately, the attention calculation of LLM inference operates on discrete sequence-level inputs that differ across requests. Here, hybrid batching does not provide data reuse.

State-of-the-art attention libraries like FlashAttention [135] and FlashInfer [26] use specialized execution for prefill and decode, attempting to maximize the compute or bandwidth utilization respectively. Hybrid batching systems serially execute these GPU kernels on the prefill and decode inputs separately, then combine them for further computation. This leads to inference switching between high compute and low bandwidth usage for prefill attention followed by high bandwidth and low compute usage for decode attention.

Our work involves computing prefill and decode attention simultaneously to maximize the utilization of both compute and memory bandwidth, and reduce the overall latency of hybrid-batch attention. For the rest of this chapter, when we refer to prefill or decode, we specifically mean the attention component of the phase, unless otherwise mentioned.

4.2.3 *Related work*

Optimizing Attention Computation. Flash-Attention [135] introduced the first specialized implementation of attention, fusing all its operations into a single kernel with tile-based computation. FA-2 [139] improved it further with better work partitioning and load balancing. FA-3 [145] improved prefill attention for NVIDIA Hopper GPUs by making use of warp specialization [146] and newer asynchronous tensor operations. FlashDecoding [25] accelerates decode attention by splitting computation along the KV dimension. FlashDecoding++ [137] uses asynchronized softmax, double-buffered flat GEMM optimizations, and dataflow-based hardware resource adaptation to accelerate decode. LeanAttention [13] follows Stream-K reduction [147] of tiled calculation to enable better load distribution across

SMs for decodes. FlashInfer [26] introduced shared-prefix based optimized attention kernels. Compared to these works that separately handle prefill and decode attention, POD-Attention jointly optimizes and fuses them into a single kernel.

Operation Fusion. Kernel fusion is a commonly used technique for improving GPU performance. Elastic kernels [148] proposes restricting resources to enable running multiple kernels concurrently. However, this method provides no guarantee of intra-SM co-location. To overcome this, ISPA [149] deploys a predetermined number of CTAs for each kernel, less than the number of CTAs that can run concurrently on the GPU. Significant a priori profiling is used to determine the appropriate CTA sizes to allow for both kernels to execute concurrently. This can be tedious for attention kernels with dynamically changing input sizes, and makes load balancing between the prefill and decode operations difficult, as one operation completing early leaves the GPU underutilized. HFuse [150] fuses kernels in warp-parallel fashion, providing source-to-source compilation tools to fuse kernels. SM-centric scheduling [151] uses the SM counter to assign work to CTAs, which POD-Attention leverages.

Optimizing LLM Inference. Optimizing LLM serving systems is an active area of research [130, 138, 127, 152, 153, 129, 128, 154, 155, 132, 156]. Orca [128] introduced iteration-level scheduling to eliminate compute fragmentation when requests of different lengths are batched together. PagedAttention [129] and vAttention [157] proposed different techniques for dynamic memory management for LLM inference. Sarathi-Serve [130] leverages chunked prefills to enable stall-free batching. In contrast, Splitwise [127], DistServe [131] and TetriInfer [158] disaggregate the prefill and decode phases onto different GPU nodes to avoid interference between these phases. Various recent works have also proposed overlapping compute with communication to improve resource utilization [159, 160, 161].

Similar to POD-Attention, NanoFlow [58] also targets improving intra-device resource utilization, by dividing a batch into smaller operation-level nano-batches and scheduling them in a way that overlaps operations with complementary resource profiles via CUDA streams. In contrast, POD-Attention tries to maximize resource-utilization within a given batch by fusing prefill and decode attention computation.

4.3 Concurrent GPU execution

In this section, we focus on GPU methods for concurrent execution, e.g., running kernels in parallel or fusing their operations into a single kernel. We both qualitatively and quantitatively analyze them, noting their performance and highlighting key limitations that motivate us to develop a specialized attention kernel. Every level of the execution hierarchy in a GPU offers potential for concurrent execution (see Table 4.1). For optimal overlap, we require that the two operations have SM-level co-location, i.e., every SM must contain both operations so that the warp scheduler can overlap their warp execution and saturate resources.

1. **Kernel-parallel.** Streams can potentially execute different GPU kernels concurrently. This approach is easy to implement as it only requires submitting existing kernels to different streams. The CTA scheduler then assigns the CTAs from the two kernels to the GPU, with the knowledge that these kernels in the two separate streams can run in parallel. All other approaches require fusing different operations into a single kernel. Unfortunately, streams alone guarantees neither concurrency nor SM-level co-location of different operations [148, 149], as the opaque CTA scheduler handles assignment of CTAs to SMs with no guarantees that an SM will contain CTAs from both kernels.
2. **CTA-parallel.** In this scheme, the two kernels are fused, and the CTAs in the combined kernel are divided across the two operations in a predetermined manner. CTA-parallel enables better load-balancing: when one CTA finishes execution, the GPU scheduler deploys the next CTA to the SM. However, similar to streams, CTA-parallel does not guarantee SM-level co-location.
3. **Warp-parallel.** Here, warps within each CTA are split across operations, as proposed in HFuse [150]. This approach guarantees SM-level co-location since every CTA contains warps from both operations and all warps in a CTA are guaranteed to reside within the same SM. Unfortunately, warp-parallel fusion suffers from the straggler

Table 4.1: Methods of concurrently executing or fusing operations along different levels of the GPU hierarchy (GC=guarantees op co-location, WQ=reduces wave quantization).

Execution method	GC	WQ	Notes
Kernel-parallel (Streams [148])	×	✓	Easiest to implement
CTA-parallel	×	✓	Easy load balancing
Warp-parallel (e.g., HFuse [150])	✓	×	Suffers from straggler problem
Intra-thread [162, 163]	✓	×	Cannot overlap with CTA barriers
SM-aware CTA (Ours)	✓	✓	Minimizes operation interference

problem: an entire CTA must complete execution before it can be replaced by another one; if one or more of its threads or warps lag behind others, the next CTA is delayed. While fusing the prefill and decode attention computation, the fused kernel requires extensive tuning to deal with a large input space of varying batch sizes and context lengths e.g., some hybrid batches may be prefill heavy and others may be decode heavy. Therefore, a fused prefill-decode attention kernel is particularly vulnerable to the straggler effect with warp-parallel fusion, e.g., decode warps within the CTA may finish early, while prefill warps lag behind. This leads to the CTA hogging SM resources that it doesn’t use.

4. **Intra-thread.** In intra-thread fusion, each thread alternates between executing instructions from different operations [162, 163]. In simple cases, this strategy provides the maximum opportunity to overlap the different operations. However, attention kernels use CTA-level synchronization barriers to coordinate fetching data into shared memory. These barriers limit intra-thread fusion as instructions before a barrier cannot be overlapped with those after the barrier, leading the operations to be serialized.

We now quantitatively analyze the performance of different methods. For complex kernels, such as attention, efficiently implementing fine-grained fusion schemes is non-trivial

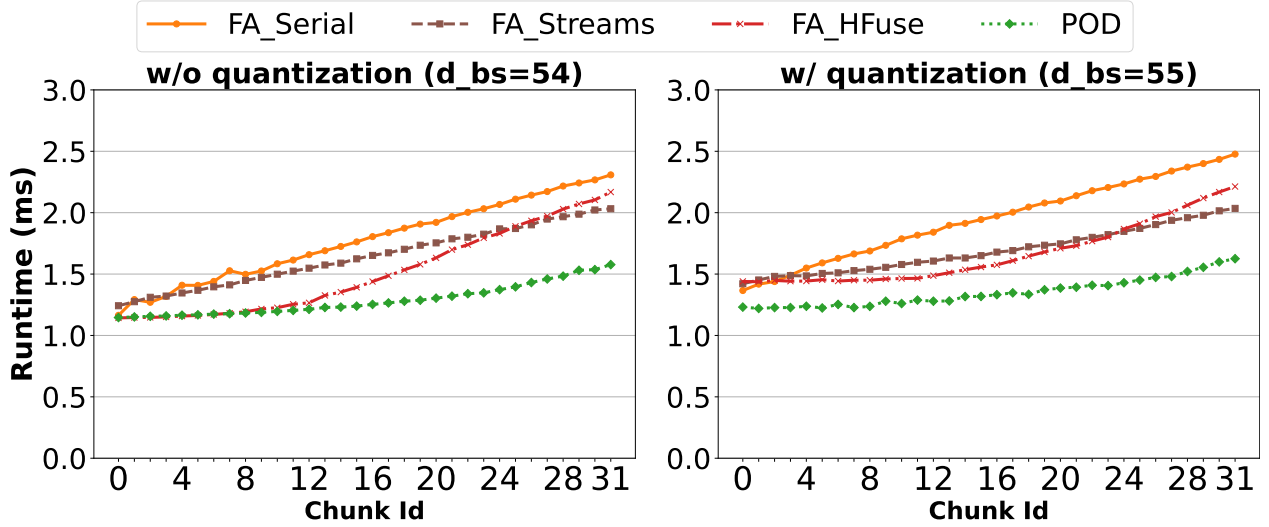


Figure 4.4: Per layer attention runtime of 32 hybrid batches corresponding to chunked prefills of a request of 16K tokens (chunk size: 512, model: Yi-6B, d_bs : decode batch size).

and prone to errors, and no readily available implementations exist for CTA-parallel and intra-thread fused attention. Therefore, we analyze kernel-parallel and warp-parallel fusion on attention kernels, then analyze the performance of other fusion methods with a simple microbenchmark.

4.3.1 Analysis of attention fusion

For kernel-parallel execution, shown as FA_Streams in Figure 4.4, we run FA’s prefill and decode kernels on two different CUDA streams. For warp-parallel execution (FA_HFuse), we fuse FA’s kernels using the toolchain provided by HFuse [150]. Figure 4.4 compares their performance against serial execution of FA’s prefill and decode attention kernels (FA_Serial). Our experiment shows the per-layer attention computation time of Yi-6B for 32 chunks of a 16K prompt (chunk size 512), each co-scheduled with decodes of 16K context length each.

Note that if the number of CTAs in a kernel is not divisible by the number of GPU SMs, some of the SMs in the last wave of scheduling can remain idle—a phenomenon known as *wave quantization* [164, 147]. In the worst case, a marginal increase in work can double the

latency of a kernel due to wave quantization. Therefore, to fully understand the benefit of concurrent execution, we evaluate performance without (left figure) and with (right figure) wave quantization. Each decode request uses 4 CTAs (one CTA per KV head). Hence a decode batch size of 54 uses 216 CTAs having no wave quantization on our NVIDIA A100 GPU (108 SMs). In contrast, a batch size of 55 uses 220 CTAs leaving 4 quantized CTAs.

FA_Streams provides some speedup over FA_Serial and its gains are higher (up to 20%) when serial execution suffers from wave quantization. This is because streams run kernels in parallel to fill GPU SMs that would otherwise remain idle. This effect can be seen in Figure 4.4 where FA_Streams take roughly the same amount of time for both batch sizes, while the time taken by FA_Serial increases at batch size 55. In particular, decode time increases by more than 25% in FA_Serial when batch size goes from 54 to 55 which increases the total attention time of prefill and decode by up to 17%. FA_HFuse outperforms FA_Streams in some cases but its performance degrades quickly due to straggler effect in the later prefill-dominated chunks. This happens because the prefill cost increases with each successive chunk but decode cost is same in all hybrid batches. Overall, FA_Streams and FA_HFuse both perform better than FA_Serial but still leave significant performance on the table, as shown by POD-Attention which outperforms both methods by a significant margin.

4.3.2 Analysis of fusion using a microbenchmark

To analyze the other methods of fusion, we construct a microbenchmark consisting of a compute-bound kernel that iteratively multiplies array elements with a scalar, and a memory-bound kernel that repeatedly adds elements of three arrays. Each thread executes a barrier after each operation. We vary the number of compute iterations to evaluate performance under varying compositions of compute-bound and memory-bound operations. Figure 4.5 shows the runtime of different fusion methods applied on these two functions. At 100 compute iterations, both operations consume equal time when executed serially. To the left of this point, memory bound is more dominant. To the right, it is compute bound. We also show

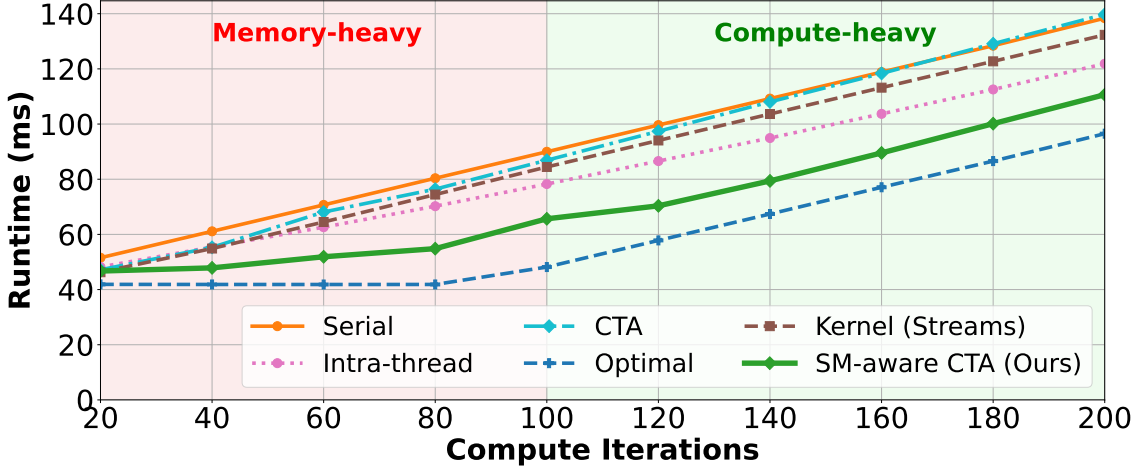


Figure 4.5: Fine-grained fusion versus serial computation.

the optimal runtime assuming perfect overlap.

CTA and kernel-parallel cannot guarantee SM-level co-location of compute-bound and memory-bound operations and hence provides only marginal average improvement of 3% and 7% over serial execution. Intra-thread fusion outperforms both serial and CTA-parallel execution, on average by 13%. However, the benefit of intra-thread fusion is limited due to sync barriers that hinder concurrent execution. Conversely, our SM-aware CTA scheduling gets much better performance and is closer to the optimal overlap.

In summary, current methods for concurrently executing heterogeneous operations face several challenges, such as straggler threads, barrier-induced delays, and the inability to guarantee SM-level co-location. In the following subsection, we demonstrate how a specialized fused kernel, designed to leverage the characteristics of prefill and decode phases, can overcome these challenges.

4.4 Design of *POD-Attention*

We introduce *POD-Attention*—a single GPU kernel that efficiently computes both prefill and decode attention. Our primary goal is to ensure that each GPU SM computes both operations simultaneously while minimizing resource contention between them. We build

our kernel atop FlashAttention v2.6.1 [139].

To achieve our goal, we fuse computation along the CTA dimension, which helps avoid the pitfalls of finer-grained warp-parallel and intra-thread fusion. In particular, CTA-parallel fusion offers three advantages: 1) it allows different CTAs to independently start and finish at different times, 2) ensures that CTA-level synchronization barriers do not affect different operations since the effect of a barrier is limited to within its CTA, and 3) it is easier to program. However, we saw that naive CTA-parallel fusion cannot guarantee that prefill and decode will be co-located within the SMs. To overcome this limitation, we introduce *software-based SM-aware CTA scheduling*, wherein each CTA decides whether to compute prefill or decode *after* it has been dispatched to an SM.

4.4.1 SM-aware CTA scheduling

SM-aware CTA scheduling co-locates prefill and decode CTAs through “runtime operation binding”. Here, a CTA decides whether to perform prefill or decode at runtime, after checking: 1) which SM it got launched on, and 2) what other CTAs running on the same SM are doing. This allows the kernel to remain completely agnostic as to how the CTA scheduler assigns CTAs to SMs, while still being able to co-locate prefill and decode.

For this, before launching the kernel, we determine how many CTAs are required for prefill and decode independently, and launch the kernel with enough CTAs matching the sum of both. Each SM has a counter keeping track of the number of CTAs launched along with 2 global counters that track the number of prefill and decode CTAs executed so far.

Figure 4.6 depicts this process. When a new CTA is assigned to an SM (shown in gray), the CTA first reads a SMID hardware register, that contains the ID of the SM it got launched on. Based on this value, it accesses the appropriate index of an SM counter array which keeps track of the number of CTAs launched on that SM so far (index 1, with value 2 in the diagram). It then increments this value to indicate a new CTA has been launched on the SM. Using the value it obtained (2 in the figure), we create a policy to decide what operation the new CTA should execute. For example, we can have even-numbered CTAs

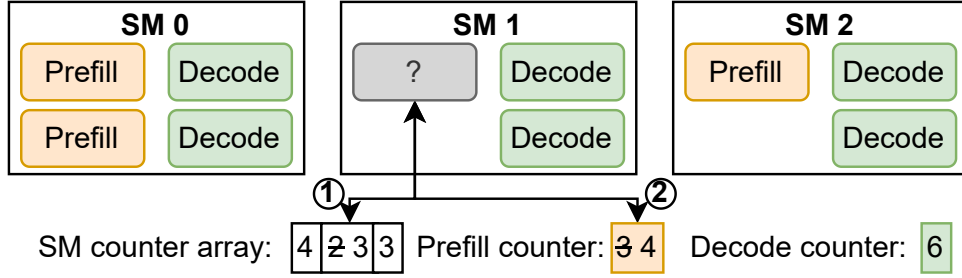


Figure 4.6: SM-aware CTA scheduling.

perform prefill, and odd decode. The CTA then increments the operation counter (prefill counter in figure), to indicate that one more of this type of CTA has executed. If the counter value exceeds the number of CTAs for the operation it switches to the other operation, e.g., if we want only 2 prefill CTAs in total and the prefill counter says 3, we perform decode instead. It then begins execution of the appropriate operation. In this way, we guarantee that both prefill and decode operations are co-located within the same SM.

4.4.2 Implementing CTA-parallel fusion

To fuse the two kernels, we first convert them into generic device functions callable from within GPU code while removing all references to the CUDA-provided CTA ID (i.e., *block-Idx*), instead passing this as a function parameter. We build a wrapper kernel that calls these different functions using a calculated CTA ID. The prefill and decode operations execute as if the supplied CTA ID was their actual ID. This enables flexible remapping of CTA IDs, e.g., CTA 0 of the fused kernel can invoke prefill with CTA ID 0, CTA 1 can call decode with ID 0, CTA 2 can call prefill with ID 1, and so on.

The amount of shared memory each CTA gets is fixed at kernel launch time, and prefill and decode operations have different requirements. To manage this, we hand-tune the shared memory usage of both prefill and decode operations to balance their requirements while minimizing performance degradation. We launch our fused kernel with enough shared memory for the maximum needed by either operation.

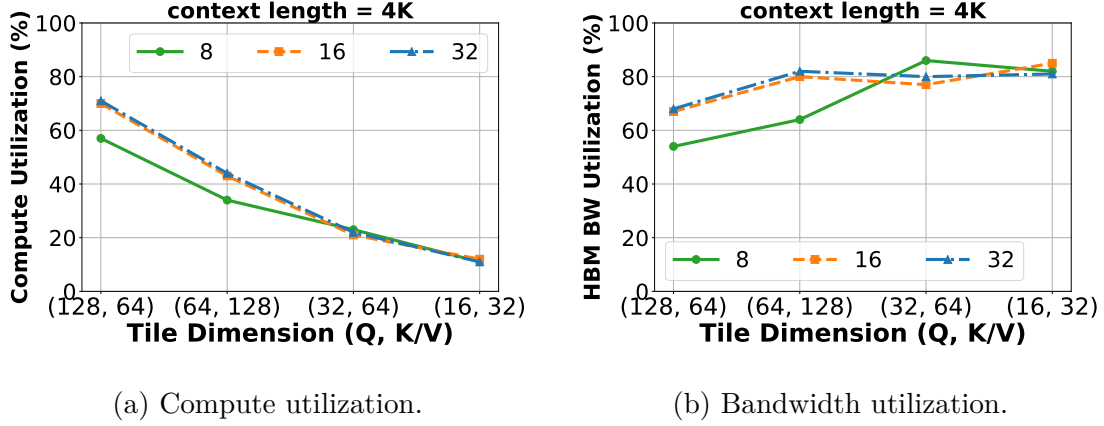


Figure 4.7: Impact of decode tile size on compute and bandwidth utilization for batch sizes 8, 16 and 32.

4.4.3 Attention optimizations

While the act of co-locating prefill and decode provides performance improvement through operation overlap, there is still scope for further optimizations.

Tile size selection: Tensor cores are designed for high-throughput matrix multiplication, taking tiles of matrices as input, and providing $8\times$ higher throughput than CUDA core counterparts [165]. Tiling is critical for data reuse; tiles are stored in the shared memory, and reused, avoiding unnecessary reads from GPU memory.

However, for decode attention, only a single token is processed per request. Group Query Attention (GQA) [166] groups multiple query heads to a smaller set of key/value heads for a model, with the group size establishing the ratio between the two. The tile length across the query sequence length (QSL) is one to eight for decode, depending on the group size of GQA. This small length makes data reuse insignificant, and is the reason that decode remains memory-bound.

By default, FlashAttention makes use of a QSL of 64 – 128 for its tiles, forcing decodes to be zero padded and creating redundant compute [137]. We evaluated the effect of the

QSL on the compute utilization of decode attention, shown in Figure 4.7a. We see that as the dimensions are reduced, the compute utilization starts proportionately reducing as well. The impact on memory bandwidth utilization was largely unaffected, shown in Figure 4.7b.

For POD-Attention, any redundant compute done by decode directly reduces the performance of the co-located prefill, as the tensor cores end up under contention. To resolve this, we make use of a smaller QSL of 16 for decode, which was the minimum that CUTLASS allows for A100 tensor operations. This improved performance in our final fused kernel.

Concurrent CTAs: The baseline FlashAttention-2 [139] kernel deploys 2 CTAs per SM, allowing us to have one prefill and one decode CTA concurrently executing. However, this limits the proportion of concurrent work to be 1:1 between the two operations. Deploying more CTAs per SM creates a tradeoff: each CTA has less resources within the SM (such as shared memory), but it provides us with opportunities to change the proportion of prefill and decode executing per SM.

Prefills prefer fewer CTAs per SM as this allows for more shared memory per CTA, and consequently supports larger tile sizes. Decodes do not benefit from larger tiles, due to their limited QSL. POD-Attention balances the two options by picking between 2 or 4 CTAs per SM. In prefill-heavy batches, POD-Attention uses 2 CTAs to provide prefill larger tile sizes. For decode-heavy batches it uses 4 to allow more fine-grained control of the proportion of prefill and decode co-location.

Virtual Decode CTAs: The amount of shared memory provided to each prefill and decode CTA must be the same in the fused kernel. However, because decode uses smaller tile sizes, the shared memory requirement of decode is a quarter of the prefill requirement. To avoid over-allocating shared memory to decodes, we divide each decode CTA into virtual CTAs containing a warp of threads. If the original decode CTA has four warps, each virtual CTA contains one warp which uses a quarter of the shared memory of the original CTA. We then modify the decode function, replacing all CTA-level barriers with warp-level barriers.

The decode function in the fused kernel is called with the appropriate virtual CTA ID, instead of the assigned CTA ID. The sum of shared memory used by all the virtual CTAs in each regular CTA is close to the shared memory used by prefill. In this way, virtual decode CTAs balance the shared memory used by prefill and decode.

Prefill Splitting: In the baseline, attention is parallelized by dividing the query tiles among the executing SMs. FlashDecoding [25] proposes splitting the computation across the KV dimension when the amount of query tiles are not enough to fill the SMs. Here, the same query tile is assigned to multiple SMs, and they operate on a discrete subset of the KV dimension. A new lightweight reduction step is introduced to merge the final computation. This improves the performance of decodes, which innately have small query dimensions and lack parallelism at small batch sizes. For chunked prefills, the query dimension is similarly small, and so splitting is done as well.

The tradeoff is that the same query tile is fetched from memory multiple times for the different SMs. This ends up increasing the memory bandwidth utilization, compared to an approach without splitting. For a prefill-only or decode-only kernel, this tradeoff is acceptable as the memory bandwidth is not saturated without increased parallelism, and so the extra memory accesses are offset by the throughput improvement from increased parallelism.

For a fused kernel, the chunked prefills and the decodes can contend on the memory bandwidth, harming fusion. To manage this, we limit the number of splits that chunked prefills can do. This embodies our *avoiding* technique for data movement, as we avoid repeated reads to query tiles without hurting correctness.

4.4.4 Discussion on alternative implementations

Concurrent execution is a well studied topic in GPU literature [167, 168, 169, 170], and our high-level goal of overlapping prefill and decode attention computation can be achieved in multiple ways. One noteworthy strategy is based on persistent threads [171, 149, 148]. In this method, one launches a pre-determined number of CTAs (enough to perfectly fill all

the SMs). Persistent threads of these CTAs pull the right type of work as necessary (e.g., prefill or decode tiles). We find that this strategy also alleviates the straggler problem. However, SM-aware scheduling is still needed to decide what operation (prefill or decode) to run on which persistent CTA, necessary for SM-level co-location. Upon integrating it with SM-aware scheduling, we find that this strategy performs on par with CTA-parallel fusion.

4.5 Evaluation

We now look at how POD-Attention performs compared to alternative approaches of attention fusion, as well as the benefit it provides in end-to-end LLM inference.

Our evaluation answers the following questions:

1. How much better is POD-Attention’s performance compared with other known methods of fusion, for attention? (§4.5.1)
2. What performance does POD-Attention provide for LLM inference? (§4.5.2)
3. How do different parameters impact POD-Attention’s performance? (§4.5.3)

Models and environment. We evaluate POD-Attention with Yi-6B [172], Llama-2-7B [173] and Llama-3-8B [174], deploying Yi-6B on one A100 GPU, and others on two A100 GPUs. Each model has 32 query heads. Each GPU has 80GB HBM memory.

Workloads and metrics. We evaluate both offline and online inference scenarios. For offline inference, we report the number of requests processed per minute. For online inference, we report TTFT, TBT and request execution latency on two workloads consisting of 2K requests each, and context lengths ranging from 4K to 32K tokens per request. One of the workloads is an internal enterprise workload (mean context length of 10.5K tokens, per-request prefill to decode token ratio, i.e., P:D in the range of 0–40) and the other is arXiv-Summarization [175] (mean context length of 9.5K tokens, P:D ratio of 0–50). On average, the number of decode tokens in the arXiv workload is 42% higher (470) than the internal workload (331).

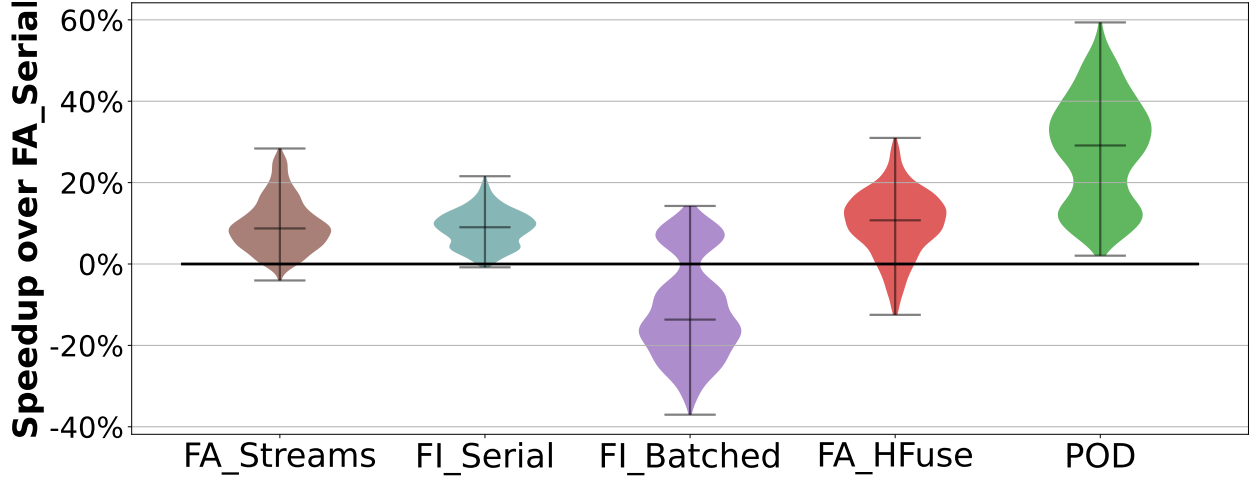


Figure 4.8: Distribution of speedup in attention computation with different mechanisms compared to FA_Serial.

Serving system baselines. Our experiments use Sarathi-Serve [144], a hybrid-batch-based LLM scheduler as the serving framework, which is built atop vLLM [176]. We evaluate two baselines: 1) the original vLLM scheduler [129] that runs prefills and decodes in separate batches, prioritizing prefills over decodes and 2) Sarathi-Serve [130] which uses hybrid batches. Both baselines use FlashAttention (v2.6.1) for attention computation. We integrate POD-Attention into Sarathi-Serve to evaluate the benefits of our optimizations. We refer to Sarathi-Serve without and with POD-Attention as Sarathi and Sarathi+POD respectively.

4.5.1 Evaluating attention computation

To demonstrate the broad applicability of POD-Attention, we conduct a comprehensive sweep across over a thousand practical hybrid batches for different models. In these experiments, we vary the context length from 4K to 20K and the prefill chunk size from 512 to 2K. We focus on scenarios where prefill and decode attention each account for at least 20% of the serial runtime, as other cases have limited potential for speedup through fusion.

In addition to FlashAttention kernels, we also compare the runtime of FlashInfer (FI) v0.2.0 kernels [26] in two configurations: FI_Serial and FI_Batched. FI_Batched com-

computes prefill and decode attention using the prefill kernel of FlashInfer. We compare against FI_Batched for two reasons: 1) this strategy is the easiest way to compute prefill and decode attention together, and 2) some systems prefer this method, e.g., Sarathi used FI_Batched in its default attention back-end [177], and a similar feature is requested in vLLM [178]. However, we show that this strategy is inefficient, e.g., when FI_Batched uses a prefill-optimized kernel, it leads to redundant compute in decode computation. This redundant computation interferes with co-running prefill. Similar interference occurs on memory-bandwidth if FI_Batched uses a decode-optimized kernel.

Figure 4.8 shows the relative speedup for different mechanisms compared to FA_Serial. FA_Streams provides limited speedup as it cannot guarantee SM-level overlap of operations. In rare cases, we find that the overhead of stream synchronization can also negate its benefits. FI_Serial has better optimized decode kernels giving it a modest improvement over FA_Serial, but it does not overlap the operations. FI_Batched improves performance at low context lengths, but degrades at higher lengths by up to 40% due to redundant computation for decodes. HFuse is the strongest baseline as it guarantees operation overlap, improving median performance by 11%. However, HFuse is susceptible to the straggler effect due to which it is slower by up to 13% compared to FA_Serial.

POD-Attention reaches a peak speedup of 59%, and a mean of 28% — higher than all alternatives. We found that in 25% of cases, it also reaches within 10% of the theoretical peak speedup, signifying near-perfect overlap. Furthermore, unlike other alternatives, POD-Attention never under-performs serial execution. These results underline the importance of a specialized attention kernel for hybrid-batching-based LLM inference.

4.5.2 Evaluating performance in LLM inference

Offline inference: For evaluating offline inference scenarios, we run long context requests of 16K tokens each. We use chunk size 512 for Yi-6B, and 1K for both Llama-2-7B and Llama-3-8B, chosen in a way that chunking a prompt does not reduce the performance of linear operations (as recommended by Sarathi [19, 130]). We run 1K total requests for Yi-6B,

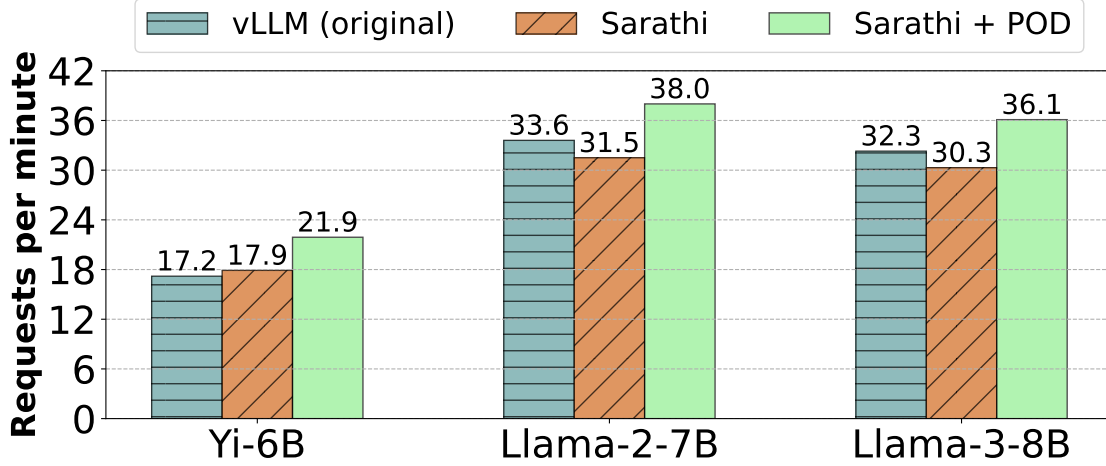


Figure 4.9: Serving throughput in offline inference.

and 2K requests each for Llama-2-7B and Llama-3-8B such that the total runtime of a single configuration is about one hour. The number of output tokens per-request is set to 2K for Yi-6B, 1K for Llama-3-8B and 256 for Llama-2-7B.

Figure 4.9 shows that Sarathi+POD delivers the best throughput: 22%, 20% and 19% higher than Sarathi, and 27%, 13% and 12% higher than vLLM, for the three models. It is worth highlighting that chunked-prefills and hybrid batching involves a tradeoff. Chunking a prompt increases attention computation time due to repeated KV cache loads: computing attention of a prefill chunk requires reading the KV cache of all prior chunks [131]. At the same time, fusing decode tokens with prefill helps execute linear operations more efficiently: model weights need not be read separately for prefill and decode. The relative performance of vLLM and Sarathi varies depending on workload, model configuration, and chunk size. In our experiments, Sarathi improves throughput slightly over vLLM for Yi-6B but underperforms it for Llama-2-7B and Llama-3-8B. Sarathi+POD fuses prefills and decodes in all operations to improve GPU resource utilization, and thereby outperforms both baselines.

Table 4.2: Internal workload. Latency numbers in seconds.

QPS	System	TTFT		TBT		Request Latency		% Requests with Stalls	
		P50	P99	P50	P99	P50	P99	200ms	500ms
1.1	vLLM (original)	0.67	10.11	0.04	1.13	25.05	91.01	99.95	97.8
	Sarathi	2.2	12.58	0.10	0.15	26.83	92.24	2.05	0
	Sarathi+POD	1.9	12.26	0.10	0.14	24.70	79.04	3.17	0
1.2	vLLM (original)	0.94	12.70	0.07	1.76	42.73	151.8	99.95	99.6
	Sarathi	25.44	57.83	0.12	0.16	67.12	140.5	5.07	2.63
	Sarathi+POD	7.49	23.78	0.11	0.15	38.69	106.8	2.29	0

Table 4.3: arXiv-based workload. Latency numbers in seconds.

QPS	System	TTFT		TBT		Request Latency		% Requests with Stalls	
		P50	P99	P50	P99	P50	P99	200ms	500ms
0.85	vLLM (original)	0.55	6.26	0.03	0.82	20.53	234.93	99.9	97.8
	Sarathi	2.68	14.89	0.08	0.13	27.87	281.07	4.15	2.05
	Sarathi+POD	1.85	12.71	0.08	0.11	24.31	255.75	1.85	1.61
0.95	vLLM	0.71	8.25	0.06	1.36	36.86	401.2	99.9	99.45
	Sarathi	46.22	144.2	0.1	0.14	90.12	417.6	4.44	1.9
	Sarathi+POD	11.74	27.38	0.09	0.12	40.6	333.0	2.2	2.1

Online inference: For online inference, the critical metrics are balancing both the TTFT and TBT latencies alongside providing a low end-to-end latency. For this, we evaluate Llama-3-8B on the internal and arXiv-based workloads near the serving capacity of the system: the maximum load a system can handle while avoiding high queuing delays [130], shown in Table 4.2 and Table 4.3. We evaluate 2048 requests in each workload by varying the input load based on a Poisson distribution. For Sarathi and Sarathi+POD, we use a chunk size of 1024 for the arXiv-based workload, and 1536 for the internal workload which is prefill heavy.

There is an inherent trade-off between the TTFT and TBT metrics [130] and optimizing for one metric can severely compromise the other. For example, as we will see below, vLLM prioritizes prefills and thus achieves low TTFT, but sacrifices TBT resulting in 95+% of user requests experiencing one or more stalls during decode generation. On the other hand, Sarathi reduces the stalls to a small % of user requests but significantly increases TTFT compared to vLLM.

	Batch size						Batch size				
CL	8	16	32	64	128	CL	8	16	32	64	128
1024	1.08	1.00	1.07	1.14	1.03	1024	1.00	1.00	1.00	1.00	1.00
2048	1.00	1.00	1.00	1.09	1.05	2048	1.10	1.06	1.05	1.00	1.00
4096	1.00	1.00	1.00	1.00	1.00	4096	1.13	1.12	1.11	1.05	1.04
8192	1.00	1.00	1.00	1.00	1.00	8192	1.13	1.15	1.16	1.11	1.08
16384	1.00	1.00	1.00	1.00	1.00	16384	1.17	1.16	1.17	1.17	1.15

(a) 2 CTAs per SM.

(b) 4 CTAs per SM.

Figure 4.10: POD-Attention with varying CTA configs.

We see that vLLM provides the lowest TTFT, as it always tries to schedule prefills as they arrive, delaying pending decodes. Sarathi increases TTFT because it batches them with ongoing decodes increasing latency. At higher loads the TTFT of Sarathi further increases due to queueing delays. Sarathi+POD is able to reduce the TTFT significantly, reducing the P99 TTFT by up to $4.3\times$.

vLLM ends up causing stalls whenever prefills arrive due to the decodes being delayed. This appears as high TBT tail latencies, impacting the interactivity of the LLM. Sarathi does not suffer from this, as its hybrid batching ensures that decodes are not delayed. Sarathi+POD further improves TBT by 10–20% by fusing attention for the hybrid batch.

We see that Sarathi+POD provides the lowest P99 request tail latency and has the fewest percentage of requests with stalls. This demonstrates how POD-Attention optimizes hybrid batching performance by overlapping the prefill and decode attention operations.

4.5.3 Sensitivity studies

CTAs per SM: Figure 4.10 shows the performance of POD-Attention with different numbers of CTAs running concurrently on an SM, varying batch sizes (horizontally) and context lengths (vertically) for Llama-3-8B. For each (context length, batch size) data point, we normalize the runtime to the best among the two configurations. In general, for long contexts

Table 4.4: Per-layer attention runtime (ms) of last four prefill chunks of a prompt, co-running with decode batch size 64 (model: Llama-3-8B, context length: 16K, chunk size: 512).

Chunk Id	FA_Serial	POD-Attention	
		Vanilla split	Limited split [Ours]
28	1.93	1.68 (0.87 $\times$)	1.45 (0.75 $\times$)
29	1.96	1.69 (0.86 $\times$)	1.45 (0.74 $\times$)
30	1.98	1.71 (0.86 $\times$)	1.45 (0.73 $\times$)
31	1.99	1.71 (0.86 $\times$)	1.46 (0.73 $\times$)

where prefill cost dominates, 2 CTAs per SM performs better as it allows for larger tile sizes. As the context length decreases, the decode cost starts dominating and hence 4 CTAs per SM starts performing better: more CTAs per SM allows packing more decodes with fewer prefills, e.g., 1 prefill CTA and 3 decode CTAs.

Scheduling policy: We explore two CTA scheduling policies within an SM, namely 50:50 allocation and Proportional allocation. In 50:50 allocation, CTAs launched on an SM alternate between prefill and decodes, i.e., the first CTA performs prefill, the next decode, and so on. This policy is agnostic to the total number of prefill and decode CTAs in the kernel. In Proportional allocation, the CTAs pick whether to perform prefill or decode depending on the total number of CTAs in the kernel. For example, if 50 prefill and 100 decode CTAs are required, the first CTA on each SM will perform prefill, the next two CTAs will perform decode, then repeat. Figure 4.11 shows the latency of POD-Attention with these policies for 8K context length and varying decode batch sizes on Yi-6B and Llama-3-8B. We notice that as the load increases (greater batch size), the performance of Proportional improves over 50:50 allocation. Proportional allocation spreads out the less frequent operations allowing better operational overlap and reduced resource contention, performing up to 14% better than a 50:50 allocation scheme.

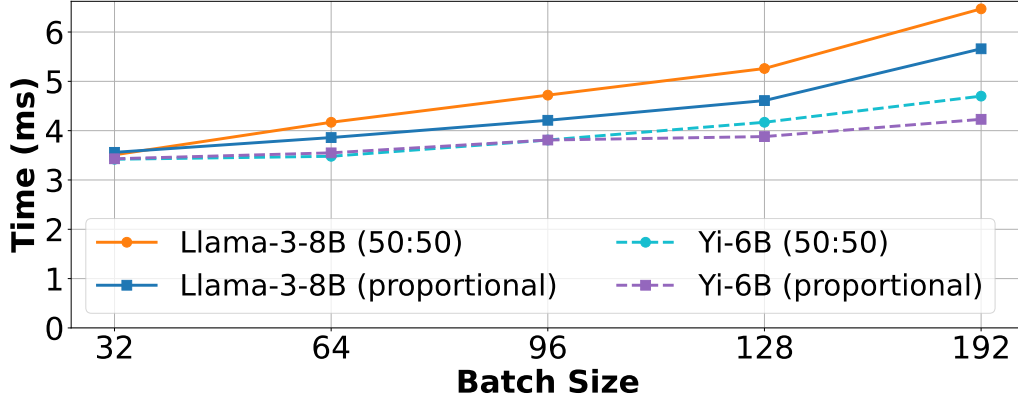


Figure 4.11: Effect of scheduling policy in POD-Attention.

Limiting prefill splits: POD-Attention reduces attention computation time with the default FlashDecoding-style splitting along the KV dimension. However, limiting the number of splits further improves performance. For example, Table 4.4 shows that in the last four chunks of a 16K prompt, co-running with 64 decode requests of the same context length, limiting the number of splits in prefill attention computation nearly doubles the speedup of POD-Attention over FA_Serial.

Sensitivity to workload: POD-Attention accelerates the execution of hybrid batches and hence its impact on overall performance depends on how many iterations consist of hybrid batches in a given workload. A workload that is highly dominated by either prefills (high P:D ratio) or decodes (low P:D ratio) is likely to experience little benefit with POD-Attention. To understand the effect of varying P:D ratio, we benchmark Llama-3-8B with a total of 2048 requests, each consisting of ≈ 16.5 K tokens, but with varying P:D ratio (in the range of 8 to 24), e.g., if the P:D is 10, then a request contains ≈ 15 K prefill tokens and ≈ 1.5 K decode tokens. Figure 4.12 shows that Sarathi+POD outperforms Sarathi over varying workload mixes. The peak gains occur in the P:D range of 12 to 18 because most batches are hybrid batches in this regime. In contrast, many iterations run decode-only batches when P:D ratio

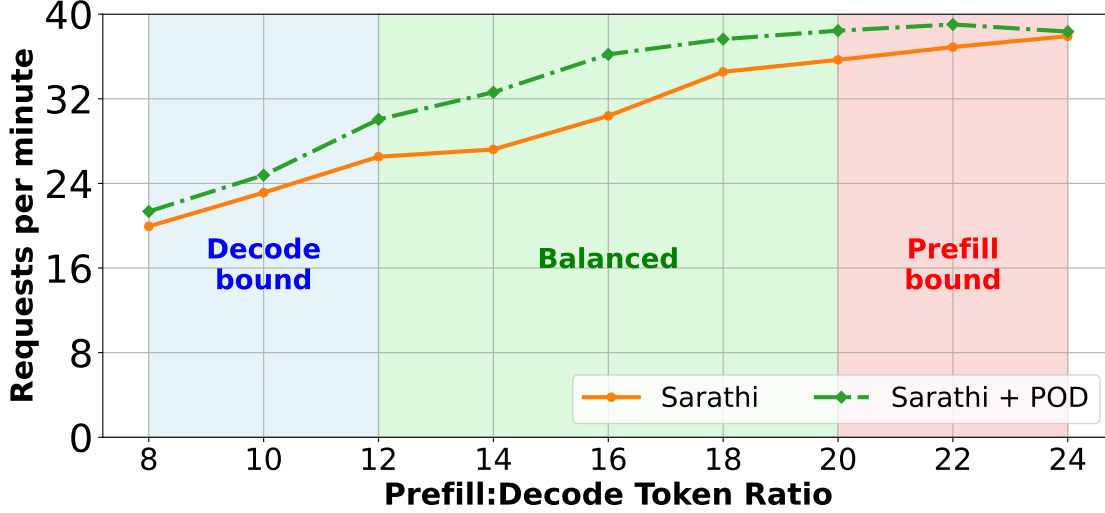


Figure 4.12: Request processing throughput under varying workload distribution (model: Llama-3-8B, TP-2).

is lower than 12 (or prefill-only batches when P:D ratio is higher than 18).

4.6 Conclusion

In this chapter we discussed POD-Attention, an instantiation of the *hiding* technique of data movement mitigations. We exploit the complementary resource profiles of LLM prefill and decode attention phases and guarantee co-location using SM-aware CTA scheduling. This allows each SM to saturate both compute units and memory bandwidth simultaneously. POD-Attention accelerates attention by up to 59% and improves end-to-end LLM inference throughput by up to 22% over state-of-the-art Sarathi. In the next chapter we will explore Invariant Bit Packing, which addresses bottlenecks at the PCIe interconnect by reducing the amount of data transferred across it, exploiting patterns within data.

Chapter 5

REDUCING THE GPU MEMORY BOTTLENECK WITH LOSSLESS COMPRESSION FOR ML

In this chapter we discuss Invariant Bit Packing (IBP), which demonstrates the *reducing* technique to mitigate data movement overheads—exploiting repetition of bits across tensors to losslessly compress the volume of data moved. This improves the performance of offloading-based ML frameworks, which store their large inputs in CPU memory. These frameworks are limited by the PCIe bandwidth connecting the GPU and CPU. Reducing the amount of data transferred directly reduces the impact of this bandwidth bottleneck.

Machine learning models such as Graph Neural Networks (GNNs), Deep Learning Recommendation Models (DLRMs), and Large Language Models (LLMs) rely on processing input tensors (e.g., vertex features for GNNs, embedding entries for DLRMs, or KV cache/weight tensors for LLMs) on the scale of hundreds of gigabytes to terabytes, far exceeding the memory capacity of any commercially available GPU. To manage this scale, data is offloaded to the CPU memory which is an order of magnitude larger than GPU memory. When needed, the input tensors are transferred across PCIe to the GPU.

This on-demand data loading causes the model execution to be bottlenecked by the PCIe bandwidth [18, 15], which is over an order of magnitude lower than the GPU memory bandwidth [1, 2, 3]. Caching [42, 20, 16] provides some benefit, but as the PCIe bandwidth is significantly lower than the GPU memory bandwidth, cache misses suffer a steep overhead. The majority of the time is spent waiting for the required data to be ferried to the GPU from the CPU. Parallelisation among compute and data transfer provides minimal benefit, as PCIe bandwidth is already saturated. Historically, GPU FLOPs/bandwidth increase faster than PCIe bandwidth, so PCIe is expected to remain the bottleneck in future [8].

Proposals to address this issue typically involve lossy compression. For example, quantization [62, 179, 180] is a popular compaction method that reduces memory footprint and data transfers. Unfortunately, lossy compression degrades model quality to an extent that is difficult to predict and varies from model to model [24]. Commercial applications often avoid sacrificing accuracy due to the risk of profit loss [181, 22].

To improve these data transfers, we investigate lossless compression as an alternative path, alleviating the PCIe bottleneck by transferring compressed tensors and decompressing them on use. Unfortunately, making GPU-based lossless compression practical for ML data transfers is a difficult task. CPU-to-GPU transfer latencies are competitive with GPU decompression latency, making traditional compression algorithms heavyweight. Further, any reliance on the host CPU in the critical path can eliminate any efficiency gains due to high-latency PCIe interactions. Finally, GPU-based decompression is complicated by the fact that common structures, such as input buffers, consume a significant fraction of GPU memory needed by ML models, making extra decompression buffers infeasible. We address these challenges by (1) choosing a host data layout that enables efficient PCIe transfers directly from the GPU, (2) performing data compression directly on host CPU memory, and (3) optimizing decompression for just-in-time use on the GPU. Unlike lossy compression, the ML model only sees uncompressed data, avoiding quality trade-offs.

We propose Invariant Bit Packing (IBP), a lossless compression technique tailored to ML acceleration that *reduces* the amount of data ferried across the PCIe. IBP analyzes the input dataset to find invariant (heavily correlated) bits across data items. These bits are stored in fixed-size metadata (order of kilobytes) within the GPU memory, then removed from the data items to compress them. IBP enables an efficient GPU implementation that utilizes asynchronous memory accesses, low-overhead warp primitives [182], and cheap bit shift operations, providing low decompression overhead and realizable throughput improvement. Our results demonstrate the effectiveness of on-the-fly decompression, improving GNN training latency by 74%, DLRM embedding lookups by 180% and LLM inference by 24% on average on an A100 GPU. Model accuracy is always preserved, as compression is completely lossless.

5.1 Background and related work

We shall now concretely look at how PCIe limits the performance of ML applications. We then go over the intricacies of CPU-GPU data transfers, followed by related work.

5.1.1 PCIe transfers in ML

Offloading is a feature offered by ML frameworks that exploits the larger CPU memory to offset the limitations of GPU memory capacity. Effective implementations rely on identifying the data to retain in the GPU memory (e.g., model weights for GNNs), and the data to be moved to the CPU memory. We now look at how specific types of models typically make use of offloading.

GNN training: GNNs are machine learning models that learn properties of graph-based data. For classification problems, GNNs use three types of inputs: a graph topology, a set of feature vectors, and a set of labels. The graph topology contains relations between nodes and edges of the graph in a sparse format. The feature vectors add more information to the graph, associated with the nodes or edges. The feature vectors can be dense floating point vectors [183] or sparse bitmask vectors [184]. The labels then group the nodes or edges of the graph. A concrete example is a citation graph, where nodes are published papers and edges are citations. Each node in the graph has a feature vector keeping track of certain keywords used in the paper. The labels finally classify each paper to its field of research. A GNN uses this information to predict labels for papers that lack them.

Modern graph datasets exceed the capacity of a GPU, and so *graph sampling* is used to construct minibatches for GNN training [185]. For each minibatch: ① a breadth-first-search (BFS) is performed, constructing a subgraph whose nodes' feature vectors are ② fetched to the GPU from the CPU for ③ GNN model training. These steps are repeated until training is complete. The fetching phase ②, where data is transferred across PCIe, is the primary focus of this work. Conversely, the GNN model weights themselves are quite small [186] and

can be held in the GPU memory.

GNNs already make use of a static cache initialized during preprocessing to reduce data movement [42, 20, 16, 187, 188]. The cache has two main components: a data store and a hashmap. All the cached data together make up the data store, possibly discontinuous or distributed across multiple interconnected GPUs. The hashmap maps a given ID to the relevant cached item. For GNNs, the node/edge ID is passed to the hashmap, which returns a pointer to the feature vector in the data store associated with the ID. Since the cache cannot store all the required data, determining what to cache remains a key challenge [42, 20, 179, 187]. Modern GNN frameworks [16, 20] use preprocessing to choose which feature vectors to cache in GPU memory.

DLRM inference: DLRMs are designed to infer facts from sparse categorical data like personal information. The model takes two types of inputs: dense contiguous features (e.g., age) and sparse categorical features (e.g., hobbies). The sparse features are transformed into dense embeddings and combined with the dense features for inference.

Each sparse feature acts as a hash key, used to look up an *embedding table*. These tables contain entries for each possible sparse feature, extending to millions of rows and exceeding GPU memory capacity. To handle the capacity requirements, tables can reside in CPU memory, with lookups moving data to the GPU [42]. Each minibatch looks up thousands of entries bottlenecking DLRM inference [189]. During inference, embedding tables are static.

LLM offloading: LLMs operate on textual data that’s been *tokenized*, i.e., converted to dense floating-point tensors (typically FP16 or BF16 data types). An operation called *attention* [136] is used to analyze the relative importance of each token with every other token. For this, each token is multiplied by a query, key, and value weight to produce a query (Q), key (K), and value (V) tensor. Every token’s Q is operated on by the other

tokens' K and V using the formula:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

This process is repeated for every new token generated by the LLM.

As every token's K and V are reused every iteration, there are significant amounts of recomputation that can be avoided. LLM serving systems manage this through KV-caching [129], which store the K and V tensors in memory when generated the first time, then reused for subsequent iterations. For large million-scale context lengths, the size of the KV cache can grow beyond the GPU memory capacity. Offloading [23] resolves this by placing the KV cache within CPU memory, transferring the tensors as needed to the GPU. This leads to the PCIe being the major bottleneck in offloaded inference. Sparse KV-caching [15] tries to reduce this impact. By using a lightweight prediction scheme, they analyze the K and V tensors to try to determine the most impactful subset. Only those tensors are then brought to the GPU. This helps reduce the bottleneck, but does not completely eliminate it, as the transfers are still along the critical path.

The LLM models themselves are quite large (order of gigabytes), possibly exceeding the capacity of a single GPU. These weights are static during LLM inference. Offloading has been used to keep the LLM weights in CPU memory, transferring them to the GPU layer-by-layer during inference [23]. This allows a single GPU to serve much larger models, reducing the hardware cost of deployment. It also allows smaller devices to serve these models.

5.1.2 CPU-GPU data transfers

A critical performance aspect for CPU-GPU data transfers is minimizing interactions with the CPU via the high-latency, bandwidth-constrained PCIe interconnect. When data in CPU memory must be transferred to the GPU, there are two methods: CPU-initiated and GPU-initiated. In CPU-initiated transfers, the CPU programs the GPU Direct Memory Access (DMA) engine to move the data to the GPU [190]. While simpler to use, for small

transfers (kilobytes) programming the DMA engine dominates the transfer time, reducing performance, as we shall see in subsection 5.4.2.

Alternatively, GPU-initiated transfers use GPU warps to directly access and move data from CPU to GPU memory through normal load/store instructions. This yields higher performance for small transfers, by eliminating DMA startup costs and avoiding a PCIe trip to the CPU to start the transfer. However, GPU-initiated PCIe transfers need to be carefully aligned to fully leverage GPU warp parallelism. Aligned memory accesses are *coalesced* into a single transaction of up to 128B by a hardware coalescer [191]. For unaligned accesses, the GPU hardware coalescer has to create multiple transactions for accesses across the alignment boundary, introducing overhead. Additionally, each PCIe transaction has header overhead, reducing effective throughput. Warps in a GPU should generate aligned, 128B transactions when accessing CPU memory [192].

5.1.3 Related work

GPU-accelerated compression. Several works propose GPU-based compression [193, 194, 195, 70, 196, 197, 198] focused on exploiting the GPU’s parallelism and high throughput to provide high compression and decompression throughput, including bit packing [193, 194]. NVIDIA GPUs A100 and newer also support lossless hardware GPU memory compression [199]. Unlike these works, IBP applies bit packing to ML datasets and CPU-GPU transfers, trading off preprocessing overheads to provide fast GPU tensor decompression.

Concurrent work has looked at lossless compression for LLMs, namely DFloat11 [200] and ZipServ [201]. These provide good compression for LLMs, but their implementations are specialized for the BF16 data type, whereas IBP generalizes to any data type.

Exploiting sparsity in ML. Many works try to reduce sparsity in ML applications. Quantization [202, 180, 203, 204, 64, 205] is a popular lossy method which trades off precision for reduced space. Compression in ML has also been investigated in the context of gradient compression [206, 207, 208, 209] and collective communication [210, 211]. Typically, compression is lossy and can affect the final evaluation results. Various works [212, 213, 214] have also

proposed hardware accelerators for GNN training, making use of hardware compression to reduce data accesses. CacheGen [215] targets the network for LLM KV transfers, proposing dynamic compression to meet latency goals under fluctuating network bandwidths.

Other sets of work [216, 217, 218, 219] look at compressed matrix algebra to improve ML workload performance. FlexPoint [220] proposes a reduced-size floating point data format, similar to TensorFloat32 [221] and BFloat16 [222] seen in commodity GPUs today. IBP instead focuses on tensor compression for commodity GPUs with limited code change to speed up CPU-GPU data transfers.

5.2 ML compression challenges

Data compression can relieve transfer bottlenecks, but is challenging to deploy in ML pipelines. Lossy compression greatly reduces data sizes and often even omits an expensive decompression step. However, it loses information. It must therefore be applied judiciously to data that can tolerate loss in fidelity without compromising application utility. Lossless compression preserves the original data, allowing it to be deployed as a black box. However, lossless decompression has overhead, complicating its adoption in performance-critical sections of an application. In this section, we describe the use of both in of ML applications. Throughout the chapter, we use two terms to describe compression performance:

$$\text{Space savings (\%)} = \left(1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}}\right) \times 100,$$

$$\text{Compression ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}$$

5.2.1 Lossy compression risks model accuracy loss

In machine learning, quantization [62] has become the major method for lossy compression. In quantization, statistical analysis is used to reduce the number of possible states that individual data elements take. For example, 16-bit to 8-bit quantization allows data to be stored in half the capacity but reduces the number of representable states by a factor of 256.

As bits are reduced, information is lost.

Employing lossy compression in machine learning comes with risks and drawbacks. As information is lost, unexpected model behavior can arise [24] and model accuracy becomes compromised. Additionally, lossy compression requires careful tuning to specific model behaviors. Choices have to be made about which data to store in compressed form and how to manage outlier data to retain model accuracy [66], which vary across machine learning models [202, 179, 64]. Both factors greatly discourage the use of lossy compression in commercial models (e.g., DLRM), where even a 0.1% loss in accuracy is considered unacceptable [22, 181].

5.2.2 *Lossless compression has high decompression overhead*

Lossless compression sidesteps these risks by guaranteeing that data can be decompressed into its original form without any loss of information. It simplifies adoption, as users do not need to worry about impacts on model accuracy, and they can treat compression as a black box for deployment. However, the high decompression overhead of lossless compression algorithms can lead to increases in latency and reduced training performance. For ML applications running on GPUs, these overheads can become pronounced.

Common code-based lossless algorithms, like Huffman Coding [223], employ decompression code tables or dictionaries that scale with input sizes to obtain high compression ratios. Input elements are mapped to bit patterns based on the frequency of element values, with more frequent values occupying fewer bits, effectively reducing data sizes. Decompression involves reading a unit of data from the input buffer and then reading the relevant code table entry/entries to map it to the output. It leads to multiple PCIe reads on the critical path for each element, degrading throughput. Further, large intermediate decompression buffers are infeasible as model inputs and/or model weights consume a significant fraction of GPU memory. Such compression algorithms are unsuitable for ML applications.

To demonstrate this, we evaluate the overhead of nvCOMP v3.0.1 [70], a closed-source NVIDIA library that provides high-performance GPU implementations of popular compression algorithms, as well as ndzip-gpu [194], an open-source GPU-accelerated compression

Table 5.1: Average speedup of compressed CPU-to-GPU transfers (best in **bold**).

Algorithm	GNN Sparse	GNN Dense	DLRM Weights	LLM KV (FP16)	LLM KV (BF16)	LLM Weights (BF16)
ANS	2.1	0.1	0.1	0.10	0.12	0.20
Bitcomp	3.9	0.4	0.5	0.65	0.65	0.64
Cascaded	8.6	0.8	0.8	0.48	0.51	0.76
GDeflate	2.1	0.3	0.5	0.16	0.16	0.18
LZ4	2.3	0.6	0.9	0.47	0.33	0.25
Snappy	1.6	0.4	0.7	0.21	0.17	0.13
zStd	1.8	0.1	0.3	0.1	0.10	0.02
ndzip	5.1	1.0	1.0	0.96	1.01	1.01
IBP	9.7	1.2	1.1	1.02	1.27	1.32

Table 5.2: Average space savings (%) across algorithms/data.

Algorithm	GNN Sparse	GNN Dense	DLRM Weights	LLM KV (FP16)	LLM KV (BF16)	LLM Weights (BF16)
ANS	76.7	-78.1	-132.6	6.2	18.84	12.40
Bitcomp	80.5	-8.8	-14.9	-0.15	0.59	-0.97
Cascaded	89.2	-1.0	-2.0	-0.01	0.72	-0.09
GDeflate	86.9	-22.7	-40.8	33.25	34.24	19.20
LZ4	89.7	-0.3	-1.0	33.33	27.63	-0.30
Snappy	88.6	-1.0	-1.0	23.24	11.64	-0.73
zStd	93.4	1.6	-2.0	38.91	43.03	21.37
ndzip	62.9	6.3	4.8	-0.21	4.95	4.31
IBP	92.9	10.4	8.3	4.49	23.43	26.71

algorithm shown to provide high decompression throughput for floating point data [224] frequently used in ML applications. Table 5.1 shows the throughput of transferring and decompressing 100,000 compressed tensors, normalized to uncompressed tensor transfer on an A100 GPU, Table 5.2 shows the space savings, and Table 5.3 shows the compression time. We evaluate LLM compression across three scenarios: KV cache in float16 (FP16) used by older models, KV cache in bfloat16 (BF16) used by modern models, and weights in BF16.

We find that many decompression algorithms perform worse than transferring uncompressed tensors, even though their space savings may be high. For example, zStd provides the best space savings for sparse GNN datasets, but its transfer speedup is one of the worst.

Table 5.3: Compression time (ms) across algorithms/data.

Algorithm	GNN Sparse	GNN Dense	DLRM Weights	LLM KV (FP16)	LLM KV (BF16)	LLM Weights (BF16)
ANS	33.1	105.4	80.4	25.2	26.8	104.6
Bitcomp	13.0	39.0	27.9	17.4	23.3	61.4
Cascaded	12.8	33.6	26.7	35.7	46.9	120.8
Gdeflate	68.5	482.7	316.8	394.2	420.9	1407.7
LZ4	135.1	69.3	43.4	105.5	141.6	495.1
Snappy	133.3	81.8	55.3	165.2	181.0	440.9
zStd	53.5	292.1	266.3	121.5	150.1	515.7
ndzip	16.4	20.7	15.0	16.5	20.6	56.2
IBP	151.5	53.2	34.2	42.8	40.9	94.9

Table 5.4: Compression of bit packing compared to the fraction of 80p-invariant bits across datasets.

Dataset	GNN Sparse			GNN Dense			DLRM Weights	LLM KV (FP16)	LLM KV (BF16)
	Pubmed	Citeseer	Cora	Reddit	Product	MAG			
Bit pack %	6.3%	6.3%	6.3%	0%	0%	0%	0%	0%	0%
Group BP %	7.5%	65.3%	76%	0%	0%	0%	0%	2%	0.1%
Invariant %	89.4%	99.0%	99.2%	15.0%	13.4%	12.9%	14.5%	8%	27%

These algorithms frequently require multiple memory accesses to reconstruct each word in the decompressed tensor (e.g., dictionary lookups in LZ4), increasing PCIe usage. Simpler compression schemes, such as run-length encoding (RLE), delta coding, and simple bit packing achieve a higher transfer speedup. *Cascaded* employs a combination of them and attains the highest speedup among existing algorithms for the sparse dataset. However, these algorithms use metadata proportional to the size of data being compressed, leading dense datasets to have compressed tensors larger than the original tensors. The compression time also varies widely across these algorithms, with *Cascaded* and *ndzip* being the quickest.

5.3 Towards practical ML data compression

To make lossless compression practical in ML workflows, it is essential to understand both the structure of input data and the architectural characteristics of GPUs. We analyze the

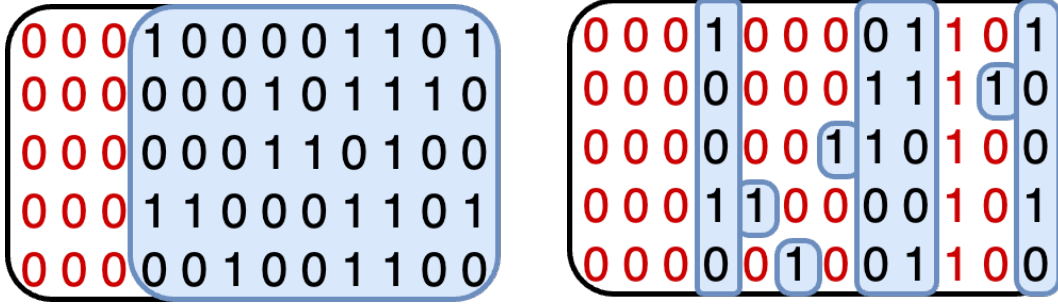


Figure 5.1: Conventional bit packing (left) and invariant bit packing (right). Red bits are removed to generate the compressed data shown in blue.

patterns of invariance found in ML data, which form the basis for our compression.

5.3.1 Invariance in ML data

From our analysis in §5.2, we find that bit packing performs best among existing algorithms. Bit packing is a family of compression algorithms commonly used in integer compression [225] to remove matching leading bits as shown in Figure 5.1 (left). Algorithms such as Cascaded [70] and ndzip [194] make use of bit packing by performing a transformation on the input (e.g., delta coding). These new data items are then packed into a smaller space by removing leading bits, e.g., 12 to 9 bits in the figure. Decompression is simple; as each compressed data item is known to be 9 bits long, offsets can be statically calculated, allowing GPU threads to parallelize decompression. As only a single 9-bit data access is needed for each 12-bit item, repeated data accesses are avoided.

5.3.2 Limits of conventional bit packing

Unfortunately, this method drastically limits the amount of compression we obtain. To highlight this, we analyzed a few publicly available datasets. Table 5.4 shows the percentage of bits that can be removed through bit packing for different datasets. As the range of values is large, the potential for space savings is very small. One possibility to boost this is to segment the data into groups and perform bit packing within these groups, as done

by ndzip and Cascaded. This works for some sparse GNN datasets (e.g., 6.3% to 65.3% for Citeseer with groups of 100 tensors). However, even for these sparse GNN datasets, compression is still limited.

5.3.3 *ML data compresses well across tensors*

We find that ML tensors tend to exhibit low entropy at the bit level, even when numerical values appear diverse. Tensors are typically drawn from structured distributions rather than random bit patterns. In floating-point representations, values for a feature vector are often bounded due to regularization and training dynamics. Exponent and sign bits frequently have the same value across many different tensors. For example, for the Reddit GNN dataset [186], we found the second to fifth most significant bits (MSBs) had the same value more than 90% of the time, and the first significant bit more than 75% of the time. These correspond to the exponent and sign bits respectively. Conversely, the other bits shared a value roughly 50% of the time, conveying a random distribution. Sparse datasets have a bias towards zero-valued bits, which we saw in the Pubmed GNN dataset [226]. Generally, when values share a global structure, certain bit positions show low variance in values.

These bit positions are low entropy, where the probability of observing the same bit value (e.g., 0) is significantly higher than 50%. We refer to these low entropy bits as *invariant*. When tensors are sampled from a stable distribution, such invariant bits remain consistent across tensors. As the number of sampled tensors increases, the empirical probability that bits are invariant converges. We can exploit this property by identifying bit positions that are consistently fixed across a large fraction of tensors and storing them once globally. The remaining bits carry the entropy and are transmitted across the PCIe normally. Thus, compression is effective whenever tensors are drawn from structured distributions with shared low-entropy bit positions, as is common in GNN features, DLRM embedding tables, and LLM KV cache/weight tensors.

We empirically demonstrate that ML data frequently has these low-entropy invariant patterns across tensors, (shown in Figure 5.1 (right)) by evaluating the invariant bits across

different ML datasets/weights; Table 5.4 shows the fraction of bits that are invariant in 80% of tensors or more (80p-invariant). We define invariance by the value identity of the same tensor bit positions. We use a threshold value T to state that a bit is invariant within an input set of tensors of cardinality N . 80p-invariant means that at least $\frac{T}{N} = 80\%$ of tensors have identical bit values in the same position. We see that all datasets have invariant bits.

Invariance also persists across data types—MAG uses FP16, LLM uses FP16 or BF16, while the others use FP32. BF16 LLM KV caches contain significantly more invariant bits compared to FP16 KV caches. BF16 uses eight exponent bits compared to FP16’s five, providing more opportunities for bits to be invariant.

To effectively reduce the amount of data ferried across PCIe, we wish to avoid transferring these invariant bits. However, this presents multiple new challenges: ① The number of invariant bits varies across tensors. This means each compressed tensor varies in size, so fixed offsets are not possible, e.g., to decompress the fifth tensor bit, we would first have to parse and decompress the first four bits. This greatly hinders parallelization opportunities. ② The bits are scattered, requiring multiple accesses to reconstruct the decompressed tensor. In Figure 5.1, each tensor requires up to four separate bit accesses to decompress.

5.4 *Invariant bit packing (IBP)*

To exploit invariance and solve these challenges, an ideal decompression technique for ML data must ① minimize the metadata required for compression, ② minimize memory accesses needed to decompress data, ③ be easily parallelize to take advantage of the GPU, and ④ support a diverse set of data types (e.g., float32, float16, bfloat16, sparse bitmasks).

We propose *Invariant Bit Packing* (IBP) to achieve these goals. IBP identifies invariant bits in data items, eliminates them, then shifts the remaining bits to reduce the space required for the data item. Decompression involves reading the compressed bits and reconstructing the original data by reinserting the removed bits. Invariant bit metadata is stored **once** in GPU memory and is used for tensor decompression **across the entire dataset**. Even for terabyte-sized datasets, the required metadata is extremely small, on the order of kilobytes.

This satisfies our first goal of minimizing the metadata for compression. The key remaining challenge is to parallelize decompression while minimizing CPU memory accesses. For GPUs, contiguous memory accesses by neighboring threads benefit from coalesced memory accesses, as the threads co-located within the same processor share a common cache. As IBP’s packing is irregular, it is hard to predict where threads should access memory for decompression.

Warp-parallel iterative decompression. We propose *warp-parallel iterative decompression* for parallelization while maintaining access locality. First, we maintain a workspace in shared memory (i.e., a partition of the L1 cache). The threads in the warp cooperate to read a contiguous segment from CPU memory to this workspace. The data in the workspace can now be decompressed. However, each thread does not know from which bit to begin reading. For this, they communicate using a warp primitive [182], which provides cycle latency communication within a warp. The threads perform a prefix scan to ascertain their starting bit offset, then independently decompress their assigned data item. Once the entire workspace is decompressed, the next segment is read from CPU memory until the entire tensor is decompressed. We use asynchronous memory accesses [227] to overlap the CPU memory access with decompression, utilizing the *hiding* technique of data movement mitigation.

As the workspace is in the L1 cache, repeated accesses during decompression do not incur any PCIe overhead. This ensures that the CPU memory is only scanned once for decompression, satisfying our second goal. Dividing a tensor across a warp also increases parallelism while minimizing communication overhead, satisfying our third goal. Finally, IBP operates at a bit level, agnostic to the underlying data type, and can be applied across a variety of input types, e.g., float32, float16, int, or sparse vectors, fulfilling our final goal.

5.4.1 Implementation of IBP

To simplify the integration of IBP in ML pipelines, we provide an easy-to-integrate open-source library. We provide a PyTorch extension [228] for Python and CUDA support through a header-only library. The Python functions are all called by the CPU, while the CUDA backend provides lower-level functions that can be called from either the CPU or the GPU.

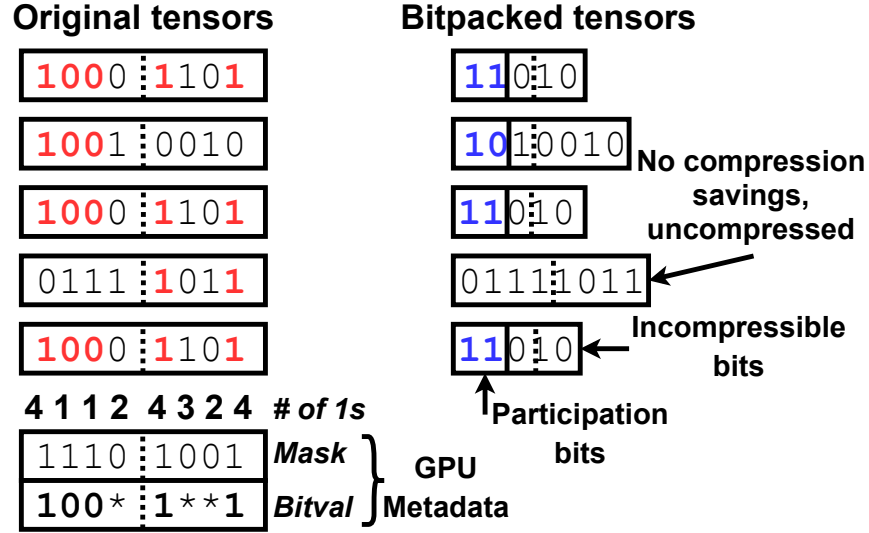


Figure 5.2: Invariant bit packing example ($T = 4$, $N = 5$).

We shall now go over the requisite steps for performing IBP in the GPU, referring to Figure 5.2 as our running example.

Preprocessing. Before we can compress the input tensors, we must determine which bits are invariant. This step is performed on the GPU to exploit its parallelism. Preprocessing is done once before model processing so that it doesn't interfere with the ML application. In our experiments, this took around three minutes for a $\sim 180\text{GB}$ dataset.

A counter is allocated for each bit across input tensors, e.g., 8 counters for 8-bit tensors, or for each vertical column in Figure 5.2. For each set bit in each input tensor, we increment the corresponding counter for the bit—the count is shown underneath the original tensors.

From this, a Mask is constructed, containing a 1 for invariant bits, and a 0 otherwise. Bitval contains the value of the invariant bits for each position. If the value of a counter is more than T , that corresponding bit is set to 1 in the Mask and Bitval. This implies that this bit is invariant with a value of 1. If the value is less than $N - T$, where N is the number of tensors in the dataset, the corresponding bit is set to 1 in the Mask and to 0 in the Bitval, implying the bit is invariant with a value 0. Otherwise, the bit in Mask is 0, implying the

bit is not invariant. For 0 bits in the Mask, the bits of Bitval do not matter, depicted with a *. We keep these bits as 0 in our implementation. For the entire dataset, only one Mask and Bitval is constructed and used, which is stored in the GPU memory due to its small size, avoiding large metadata overheads that scale with dataset sizes.

The value of T has a heavy impact on the achieved compression ratio. If T is too high, fewer bits get chosen as invariant, limiting the maximum achievable compression. If T is too low, excessive bits are chosen as invariant, leading to mismatches between tensors and Bitval values causing many input tensors to remain uncompressed. To maximize compression, we perform a sweep over T values from $0.7N$ to N , picking the value providing the greatest observed compression. In practice, $T = 0.8N$ strikes a good balance that maximizes compression for our datasets.

Compression. Using Mask and Bitval, we compress the tensors. Each tensor is first broken into fixed-size chunks (4 bits in Figure 5.2), allowing us to parallelize compression (and later, decompression) across chunks, with each chunk handled by a different GPU thread. A metadata bit is prepended to the compressed tensor for each chunk in the original tensor, called the *participation bit* (blue in Figure 5.2). The participation bit indicates whether that chunk is compressed. A chunk can be compressed if its masked bit values match the Bitval — formally if $Chunk_i \wedge Mask_i = Bitval_i$, for chunk i . For chunks with matching values, the matching bits are removed in the compressed tensor (e.g., 100 for the first chunk in the figure), and the corresponding participation bit is set to denote that the chunk is compressed. Otherwise, the participation bit is unset, e.g., the second chunk of the second tensor.

Similar to preprocessing, compression is performed by the GPU for parallelism. The tensors are kept within high-capacity CPU memory and accessed by the GPU using CUDA’s zero-copy Unified Virtual Memory [229], allowing the GPU direct access to CPU memory.

Figure 5.3 shows the pseudo-code of the two main functions used for compression: precalculating a tensor’s compressed size and the actual compression. To calculate the compressed size of a tensor, we partition the set of all input tensors among the GPU warps, with each warp handling a subset of the input set. Each thread in the warp, in turn, is handling a

```

1  def getCompressSize(tensor, tensorLen, Mask, Value):
2      bitsSaved = 0
3      i = THREAD_ID
4      while i < tensorLen / CHUNK_SZ:
5          chunk = tensor[i * CHUNK_SZ]
6          maskChunk = Mask[i * CHUNK_SZ]
7          bitvalChunk = Value[i * CHUNK_SZ]
8          if (chunk BITAND maskChunk) == bitvalChunk:
9              bitsSaved += popcount(maskChunk)
10             i += WARP_SZ
11     bitsSaved = warpScan(bitsSaved)
12     # See if tensor is worth compressing
13     if THREAD_ID == 0:
14         participationBits = tensorLen / CHUNK_SZ
15         bytesSaved = (bitsSaved - participationBits) / 8
16         if bytesSaved <= 0:
17             return tensorLen
18         else:
19             return tensorLen - bytesSaved
20
21 def compress(tensor, tensorLen, Mask, Value):
22     j = THREAD_ID
23     bitShift = 0
24     while j < tensorLen / CHUNK_SZ:
25         curBitshift = 32
26         chunk = tensor[j * CHUNK_SZ]
27         maskChunk = Mask[j * CHUNK_SZ]
28         bitvalChunk = Value[j * CHUNK_SZ]
29         if (chunk AND maskChunk) == bitvalChunk:
30             curBitshift -= popcount(maskChunk)
31             bitshift += warpScan(curBitshift)
32             insertBits(outputTensor, chunk, bitshift)
33             j += WARP_SZ
34     return outputTensor

```

Figure 5.3: Pseudocode for IBP tensor compression.

separate chunk of its input tensor. The warps execute in parallel. For simplicity, we examine the operation of a single warp.

Each thread checks whether the bitwise AND operation between the chunk and *Mask* matches the *Bitval*, implying that the chunk can be compressed [lines 5–8]. If so, the number of set bits in the *Mask* indicates the number of bits saved in the chunk [line 9]. This process is repeated for the entire input tensor [lines 4, 10]. The warp then performs an in-place scan operation to determine the combined bits saved [line 11]. The leader thread of the warp [line 13] performs the final calculation of the compressed size. The compressed tensor requires an extra bit per chunk for the participation bits, which are subtracted from the bits saved [lines 14–15]. We keep all the tensors byte-aligned to avoid complex sub-byte memory accesses. Hence, we can divide the bits saved by 8 to get the bytes saved. If we find no bytes are saved, we keep the tensor in uncompressed form, without participation bits, returning the original size (e.g., the second-to-last tensor in the figure). In this way, the compressed dataset cannot exceed the size of the uncompressed dataset. Otherwise, we return the new compressed size [line 18]. For the input tensors with compressed sizes smaller than the tensor size, we perform compression [line 21].

To compress the input chunk into the output buffer, each thread needs to know where to insert the compressed chunk. For this, we calculate the *bit shift* of the current chunk or the bit index at which the current chunk will be inserted. This value depends on how many bits the previous chunks inserted into the output tensor, which are being compressed in parallel by other threads. Waiting for them to complete serializes the operation, reducing compression performance.

To resolve this, we make use of *warp intrinsics* [182], which is a GPU feature that allows threads within a warp to communicate with each other with low latency (order of cycles). Each thread first calculates the number of bits that its assigned chunk will occupy in the output tensor [lines 26–30], and then all the threads communicate this value with each other using the warp intrinsic [line 31]. The threads then insert their chunk into the output array independently [line 32], maximizing parallelism.

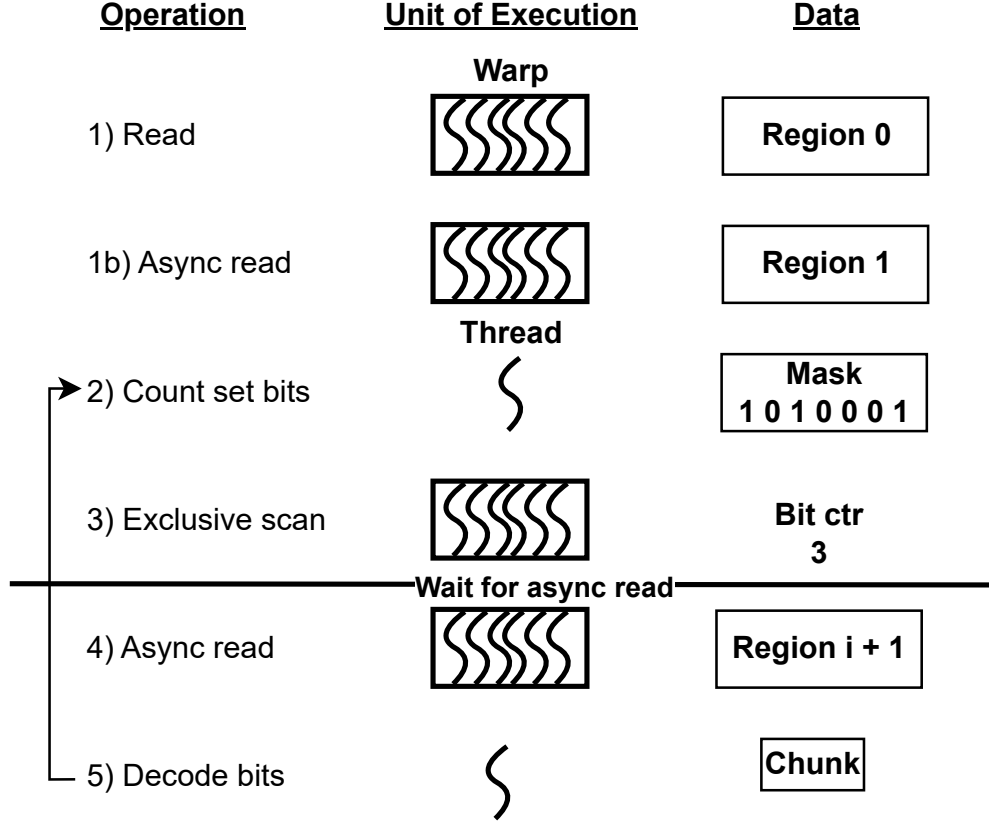


Figure 5.4: IBP tensor decompression steps.

Decompression. Decompression is on the critical path of performance and must have minimal overhead. To attain this, we rely on lightweight warp primitives, bit shift operations, and asynchronous CPU memory accesses. We decompress input tensors in batches, with each tensor being decompressed by a separate warp, avoiding communication outside the warp and reducing synchronization overheads.

Figure 5.4 shows the major steps involved in decompression. Each thread in the warp is responsible for decompressing a chunk within the tensor. 1) The warp collaboratively reads a region of data from the CPU memory to the workspace. This collaborative read allows the operation to benefit from memory coalescing (subsection 5.1.2). Following this, an asynchronous read is initiated proactively (1b).

2) Each thread then reads its respective chunk of the Mask metadata and counts the number of unset bits. This indicates to the thread the number of bits that it will need to fetch from the shared workspace. 3) The warp then performs an exclusive scan on these bit counters using warp primitives to obtain individual offsets. Each thread in the warp is now aware of the bits the threads before it intend to insert, yielding the start offset for itself.

The warp then waits for the prior asynchronous read to complete. 4) If further CPU reads seem necessary, another asynchronous read is kicked off for the next region. 5) Each thread then performs the required bit reads and bit shifts from the workspace to decompress their assigned chunk. This process is repeated until the entire tensor is decompressed.

At all points, communication is contained within the warp, reducing synchronization overheads. CPU memory accesses are also asynchronous allowing the decompression logic to overlap with data movement.

5.4.2 Efficient PCIe data transfer

Efficient transfer of data across PCIe is crucial to reducing the overhead of decompression. In Figure 5.5, we evaluate four possible methods of copying randomly distributed, unaligned array elements of different sizes from CPU memory to a GPU buffer, along with the hardware limit. The first two are CPU-initiated. *CPU (Fine-grained)* copies element-by-element to the GPU buffer. This performs the worst, as a DMA copy is performed for each array element, which has startup overheads. *CPU (Bulk)* copies individual elements to a contiguous CPU buffer, then copies the CPU buffer at once to the GPU buffer. This performs better, as the DMA copy happens only once, amortizing the startup overhead. *GPU* uses the CUDA zero-copy feature [229] to perform copying from within the GPU. This leverages the GPU’s parallelism to transfer the array elements in parallel performing better.

From this investigation, it is clear that initiating copies from the GPU performs better. However, as discussed in subsection 5.1.2, 128B size and aligned transfers provide maximum link utilization. Unfortunately, compressed tensors are misaligned and oddly-sized, limiting performance. For this, we use two techniques: ① aligned and ② bounded data transfers.

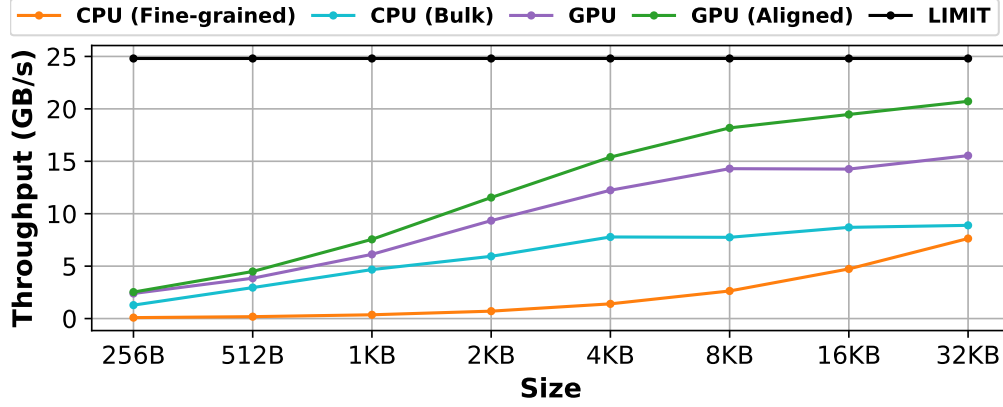
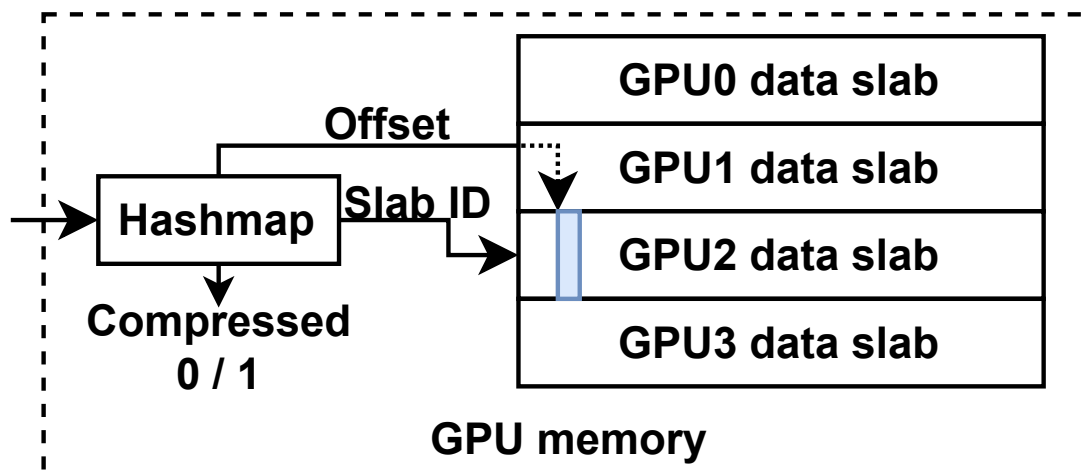


Figure 5.5: CPU-to-GPU copy throughput across methods.

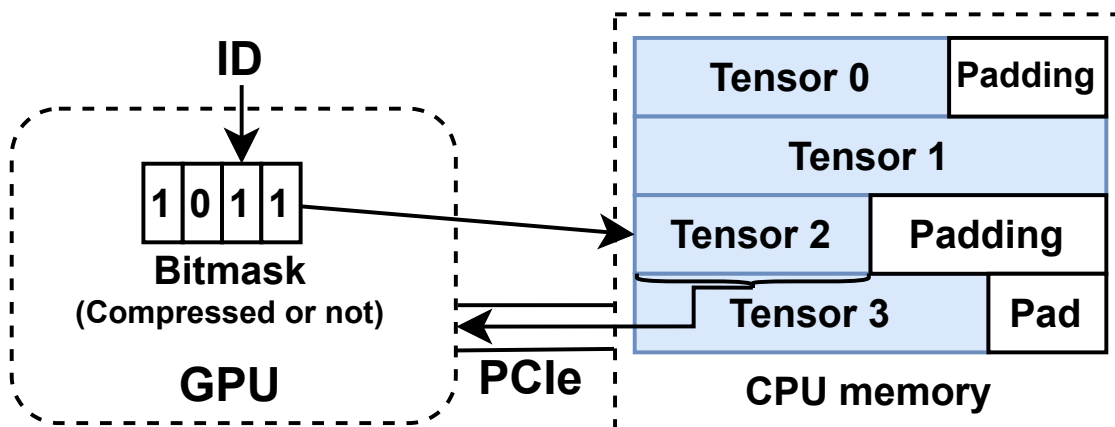
Aligned transfers. All our data transfers use aligned copies, where we find the 128B boundary for a CPU buffer being copied, and shift indices so that each warp starts at a 128B-aligned boundary. We proceed with the accesses, copying the required data into a GPU buffer, with extra PCIe transactions only to copy unaligned portions. *GPU (Aligned)* shows this improves throughput by 25% over simple GPU copying.

Bounded transfers. Our second technique, bounded data transfers, uses a lower bound and upper bound instead of a fixed length when copying. This guarantees aligned transfers when iteratively reading from a contiguous buffer. The function uses a warp to copy 128B aligned segments from a source to destination, stopping when it has copied at least the lower bound up to the upper bound of data.

By ensuring the lower and upper bounds are separated by 128B, we can copy data into an intermediate buffer, operate on it, and then copy more data while always copying 128B-aligned segments. For example, Step 1 of Figure 5.4 copies between 1–128 bytes based on the alignment of `src`. Without bounded transfers, if the source was misaligned and we repeatedly copied 128B into a buffer, every copy would be misaligned.



(a) Compressed cache.



(b) Compressed CPU memory.

Figure 5.6: Integrating IBP into ML applications.

5.5 Integrating IBP into ML frameworks

A key aspect of IBP’s design is making integration into frameworks simple, requiring modifications only at data transfer boundaries and avoiding a full application rewrite. To this end, we now see how we provide cache and PCIe transfer compression to these frameworks.

Cache compression: Frameworks sometimes use a static cache in GPU memory to store frequently accessed data [16, 42]. We enhance this cache by storing tensors in compressed format and decompressing on use, increasing the cache’s effective capacity, shown in Figure 5.6a. The cache typically has two software components, a datastore, where the tensors are stored, and a hashmap, which maps the tensor ID to the tensor location in the datastore.

To enhance a static GPU cache with compression, we must pack compressed tensors into the data store, adding new mappings into the hashmap. To achieve this, we change the data store layout and hashmap output.

First, we allocate a contiguous memory region per interconnected GPU, which we call a data slab; e.g., an 8-GPU system would have 8 separate data slabs. Together, these slabs make up the data store. We compress the input tensors and lay them contiguously within the data slabs, keeping track of the offset within the slab and the GPU ID of the slab. In this way, our compressed cache consumes the same space as the uncompressed version.

During cache access, we require two pieces of information: where the tensor is located and whether it is in compressed or uncompressed form. To avoid additional data structures or memory accesses, we integrate this information into the existing hashmap structure. Instead of a 64-bit pointer to the relevant tensor, the hashmap now maps an ID to a 3-bit data slab ID, a 40-bit offset, and a bit indicating whether the tensor is in compressed form or not. The 3-bit slab ID allows us to support up to 8 GPU slabs (the current multi-GPU limit), and the 40-bit offset allows us to index up to 1 TB of GPU memory, exceeding current GPU capacities. We have 20 spare bits in case we need to extend these in the future.

Using the data slab ID, we identify which GPU slab the required tensor is on and then

use the offset to index into the appropriate location. The 1-bit compressed indicator then lets us know whether to decompress the data or simply read it. This retains all the existing features of the static cache, while seamlessly introducing compression.

PCIe transfer compression: While cache compression improves the hit rate of the cache, misses are still fetched from CPU memory. To reduce PCIe transfer overhead, we also compress CPU tensors. Two difficulties arise from this. (1) For most datasets, input tensors are fixed in size and accessing a specific tensor involves shifting a pointer by a fixed offset. For example, to access the fifth tensor, one simply shifts 5 tensor indices. The sizes of compressed tensors vary, and we cannot use a fixed shift to index a tensor. Tracking offsets adds extra memory accesses when decompressing and increases memory usage. (2) IBP retains certain tensors in uncompressed form, which must be separately tracked.

To solve the first problem, we perform in-place compression, where the entire dataset occupies the same amount of memory, but each individual tensor is compressed. For example, a 100B tensor may be compressed to 80B. The full 100B are allocated, but only the first 80B are useful. When reading tensors from the CPU, only the required data is read, with individual tensors indexed as before. The CPU memory is many times larger than the GPU’s, making this acceptable.

To solve the second problem, we maintain a bitmask of compressed and uncompressed tensors in GPU memory, which allows IBP to look up whether a tensor is compressed, and if so, decompress. Otherwise it simply transfers the entire tensor. Each bit represents whether the CPU tensor is compressed or not. Before reading a tensor from the CPU, this bitmask is first checked. This has a memory overhead of 1 bit per tensor, or $Bitmask\ size = Dataset\ size / (Tensor\ size * 8)$. For a 512 GB dataset with 1 KB tensors, this bitmask occupies only 64 MB, fitting well within GPU memory.

5.6 Evaluation

For our evaluation, we answer the following:

1. How well does IBP compress ML datasets? (§5.6.1)
2. What performance improvement does IBP provide GNN training? (§5.6.2)
3. What performance improvement does IBP provide DLRM transfers? (§5.6.3)
4. What performance improvement does IBP provide offloaded LLM inference? (§5.6.4)
5. How do IBP’s parameters impact performance? (§5.6.5)

System configuration. We evaluate IBP on a system with an A100 GPU with 300 GB of CPU memory. We use CUDA 11.7, Python 3.8, and Pytorch 1.13.1. The system has a 16-lane PCIe Gen4 connection between the CPU and GPU, providing a theoretical bandwidth of 32 GB/s.

Baselines. For compression benchmarking, we use nvCOMP v3.0.1 [70], a closed-source NVIDIA library that provides GPU compression algorithms, as well as ndzip-gpu [194], an open-source GPU compression algorithm for floating point data. We integrate IBP into two GNN frameworks: DGLv2.1 [230], a popular graph library, and Legion [16], a state-of-the-art GNN framework. For DLRM, we integrate IBP into Colossal-AI’s CachedEmbedding-Bag [231, 203]. For LLM, we add IBP support to FlexGen [23], where we offload LLM weights, and InfiniGen [15], where we offload the KV cache. We compare IBP against native applications without compression.

5.6.1 IBP compression performance

Table 5.1 shows that IBP provides the best transfer throughput. We now take a deeper look at the compression ratio that IBP provides and compare it to alternatives, shown in Table 5.2. The sparse GNN datasets use sparse tensors. For these, most algorithms are able to provide good compression, with zStd providing the best. The dense GNN, DLRM weights, and LLM datasets consist of dense floating-point numbers that are harder to compress. zStd and ndzip

can compress some of these, but IBP is the only algorithm that compresses all of them, giving 10–12% space savings for dense GNN datasets, 8% for DLRM, and up to 27% for LLM.

We experiment with different forms of LLM compression. We first compressed the KV cache generated by an OPT-30B [232] model, which yielded only 4–5% space compression. This model uses the FP16 data type for KV cache entries. Modern models prefer the BF16 data type, which uses eight bits for the exponent compared to the five exponent bits of FP16. The expanded exponent allows the values to take a wider range, improving LLM accuracy. As we note in subsection 5.3.1, exponent bits tend to be invariant, as values across tensors end up in similar ranges.

We test the Gemma-7B [233] model which uses the BF16 type. IBP delivers significant compression for both the LLM KV cache and weights for this model at 23% and 27% respectively. This benefit came with *no changes* to IBP’s algorithm, giving a strong demonstration of how it generalizes across data types and models. Overall, IBP’s compression across all datasets translates to practical throughput gains.

We also investigate the compression time in Table 5.3. For IBP, this includes the time taken to preprocess the dataset to generate the *Mask* and *Bitval*. IBP requires a moderate amount of time to compress; we find that around 50–70% of the time was spent on preprocessing the dataset, with the rest being shifting bits to compress the tensors. Generally, preprocessing is done off the critical path, reducing the significance of this overhead.

Decompression overhead. We evaluate IBP’s decompression overhead versus space savings across different tensor sizes via a microbenchmark. To achieve specific space savings, we create synthetic *Mask* and *Bitval* metadata, varying the set bits in the *Mask* to yield the desired compression ratio. We compress 100,000 tensors of sizes representative of ML datasets (256B, 1KB, and 4KB) in CPU memory using the appropriate metadata and measure the total time taken to transfer and decompress the compressed tensors, averaged across 100 iterations. To highlight decompression overheads, we normalize the throughput to that of a mock system that transfers compressed tensors across PCIe, but writes uncompressed tensors to GPU memory without decompression.

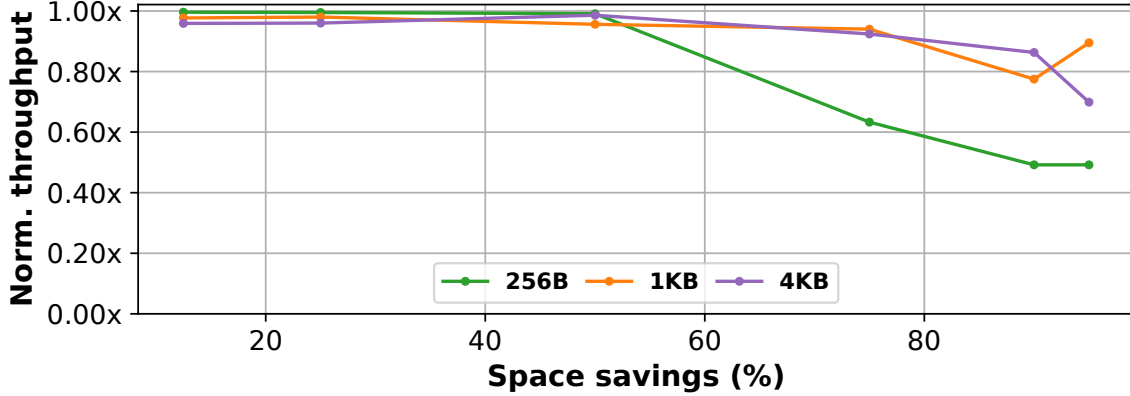


Figure 5.7: Decompression throughput versus space savings.

Table 5.5: Compression ratios with different fractions of dataset sampled during preprocessing.

Dataset	1%	5%	10%	25%	50%	100%
Pubmed	7.37x	7.34x	7.34x	7.34x	7.35x	7.35x
Citeseer	25.09x	25.09x	25.09x	25.09x	25.09x	25.09x
Cora	26.36x	26.36x	26.36x	26.36x	26.36x	26.34x
Reddit	1.13x	1.13x	1.13x	1.14x	1.14x	1.14x
Products	1.12x	1.12x	1.12x	1.12x	1.12x	1.12x
MAG	1.11x	1.11x	1.11x	1.11x	1.11x	1.11x
DLRM	1.11x	1.12x	1.13x	1.13x	1.13x	1.14x
LLM KV (FP16)	1.03x	1.04x	1.06x	1.05x	1.05x	1.05x

Figure 5.7 shows that for space savings below 50%, which covers the dense ML datasets, the decompression throughput is above 95% of ideal. PCIe is the bottleneck in these cases, and the decompression overhead is hidden by the GPU’s parallelism and asynchronous memory accesses. For compressed sizes smaller than this, i.e., 256B with 50% or more compression, overhead is seen as excess data is brought in with a single 128B transfer. As the space saved increases, throughput starts reducing as the overhead of decompression (e.g., GPU accesses, bit shifts) has an increasing impact on performance. At 90% space saved, which covers sparse datasets, throughput is at 78% and 86% of ideal for 1KB and 4KB, respectively, showing low overhead. Only small, 256B tensors suffer from an overhead of 50%.

Table 5.6: Datasets used for GNN training. *SU are scaled up.

Dataset	PubmedSU	CiteseerSU	CoraSU	Reddit	Products	MAG
Size per tensor	2KB	14.5KB	34KB	2.4KB	400B	1.5KB
Total tensor size	4.6GB	34GB	79.5GB	0.52GB	0.91GB	174GB
No. of nodes	2.4M	2.4M	2.4M	233K	2.4M	121M
No. of edges	123M	123M	123M	114M	123M	1.3B
Dataset size	5.1GB	34.5GB	80GB	0.95GB	1.4GB	180.2GB
Batch size	8,192	1,024	512	1,024	8,192	8,192

Dataset Sampling. It may be the case that the entire dataset cannot be preprocessed together, e.g., in distributed deployments, when data is updated at runtime, or streamed in online scenarios. For this, we can sample a fraction of the dataset for preprocessing.

In Table 5.5, we generated a *Mask* and *Bitval* using a specified fraction of data instead of the entire dataset. We then evaluated the compression ratio obtained using this *Mask* and *Bitval* for the entire dataset. This allows us to see how well a sampled *Mask* and *Bitval* generalizes to all data. We see that samples generalize well, constructing good *Mask* and *Bitval* for our datasets. Even when 10% of the dataset is sampled, the compression ratio obtained is comparable to the entire dataset in most cases.

5.6.2 GNN training speedup

Datasets. We evaluate 6 publicly available GNN datasets: Pubmed [226], Citeseer [184], Cora [234], Reddit [186], Products [183] and papers of MAG240M [183], spanning various tensor and dataset sizes (Table 5.6).

GNN training is commercially performed on datasets that span 100s of GBs to TBs in size [21]. Unfortunately, publicly available datasets are significantly smaller. To compensate, we perform two modifications: 1) Pubmed, Citeseer, and Cora are very small datasets (order of MB); we substituted the original graph topology with the topology of Products to scale

up the size of these datasets. Input tensors from the original dataset were randomly assigned to the vertices of this larger graph. We marked these datasets with the suffix *SU* to indicate that they have been scaled up. This retains a real-world graph topology, similar to scaled-up studies performed by prior work [16, 188]. As the tensors themselves are unmodified, scaling up does not affect the compression ratios obtained, which we empirically validated. 2) To imitate cache hit rates of large-scale training [21], we restrict the GPU cache size to 1% of the graph’s total input tensor size.

Frameworks. We extend DGL with our PyTorch extension to compress input tensors when transferring them from the CPU to the GPU. Users call `compress` once during graph initialization to enable IBP. IBP’s one-time preprocessing and compression took between half a second to 3 minutes, depending on the dataset size. After this, DGL’s sampling process automatically decompresses input tensors when transferring them to the GPU. Adding IBP to DGL required less than 35 lines of code, showing the simplicity of our Python API.

Legion is primarily written in CUDA and uses parallel processes for graph sampling and neural network training to overlap the two tasks. It additionally makes use of a software-based static cache within the GPU memory to cache input tensors used for training and graph topology structures used for graph sampling.

We extend Legion by compressing the static cache to fit more input tensors within the same space. We also compress the input tensors residing in CPU memory, decompressing them after transferring to reduce PCIe latency.

Setup. We run the GraphSAGE model [186] using two-hop random neighbor sampling with a fanout of 25 and 10. We attempted to set a batch size of 8K used by prior works [16, 188], but ran into out-of-memory errors for certain datasets due to our feature tensors being larger than in those works; the evaluated batch sizes of datasets are shown in Table 5.6. We run 10 epochs of training for each dataset, and average the runtime of the last 9 epochs to avoid startup overheads seen in the first epoch. IBP uses 4B chunk sizes as this provided good compression, while many of the dataset tensors are not divisible by an 8B chunk size. We verified that IBP’s lossless compression doesn’t impact model accuracy.

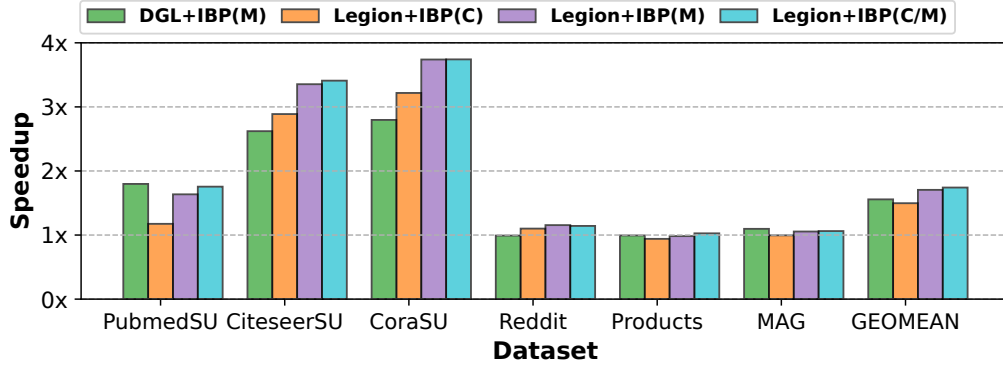


Figure 5.8: Average GNN training epoch speedup.

Table 5.7: GNN tensor cache capacity with/out compression.

Cache	PubmedSU	CiteseerSU	CoraSU	Reddit	Products	MAG
Original	24,401	24,401	24,401	2,301	24,401	1,217,501
Compress	166,901	589,182	631,609	2,596	26,987	1,354,043
Increase	6.84×	24.15×	25.88×	1.13×	1.11×	1.11×

Performance. Figure 5.8 shows the speedup of IBP over DGL and Legion, normalized to the respective framework. IBP(M) compresses the traffic across the PCIe, IBP(C) compresses the static cache, while IBP(C/M) does both. On average, IBP speeds up DGL by 56% and Legion by 74% on an A100. The biggest gains IBP sees on top of Legion are due to PCIe traffic compression, improving throughput by up to $2.7\times$, and on average 70%. In general, having both cache compression and PCIe compression combined improves speedup over having them individually.

We also look at how compression influenced the in-GPU tensor cache used by Legion in Table 5.7. We see that the increase in capacity of the cache largely correlates with the ratio of compression of the datasets (Table 5.5). More compressible datasets increase the capacity of the cache proportionately.

Latency breakdown. We examined the sources of latency in Legion, with and without IBP, for the CoraSU and Reddit datasets. We show these in Figure 5.9. Legion uses two

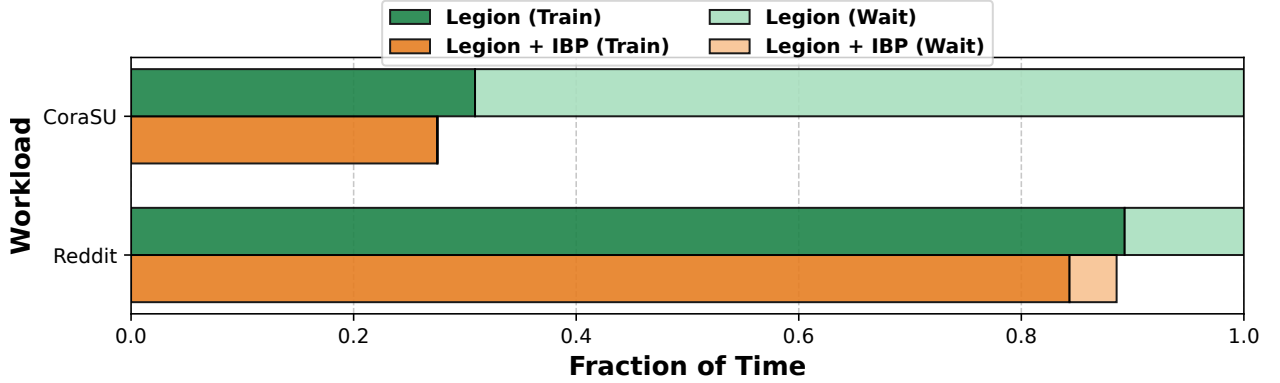


Figure 5.9: Breakdown of latency for GNN training epochs.

processes: one for the computationally-heavy training operations, while the other samples the graph, prepares input, and fetches it into GPU memory. These are done in parallel on two separate minibatches. The training process trains on a minibatch, while the graph sampler prepares the next minibatch. We measured the time the training process spends training (Train) and waiting for the next batch (Wait). Wait shows the exposed latency of the sampler not hidden by Train.

We see that IBP reduces the Wait time greatly by reducing the amount of time spent transferring data. For CoraSU, Wait time is almost eliminated entirely. Interestingly, even the Train time slightly reduces. Legion performs data transfers using a subset of available SMs, with the remainder used for training. IBP reduces data transfer volume, leading to a net reduction in transfer time. Consequently, transfer SMs are free for ML training earlier, causing a cascading effect speeding up Train time. For CoraSU, IBP decreased training time by 12% versus Legion, due to reduced SM contention for data transfer.

5.6.3 DLRM inference speedup

As discussed in subsection 5.1.1, large embedding table lookups and transfers are the bottleneck of DLRM inference [189, 235]. Hence, we evaluate the benefit of IBP on embedding lookups. Besides bookkeeping, adding IBP support required less than 15 lines of Python

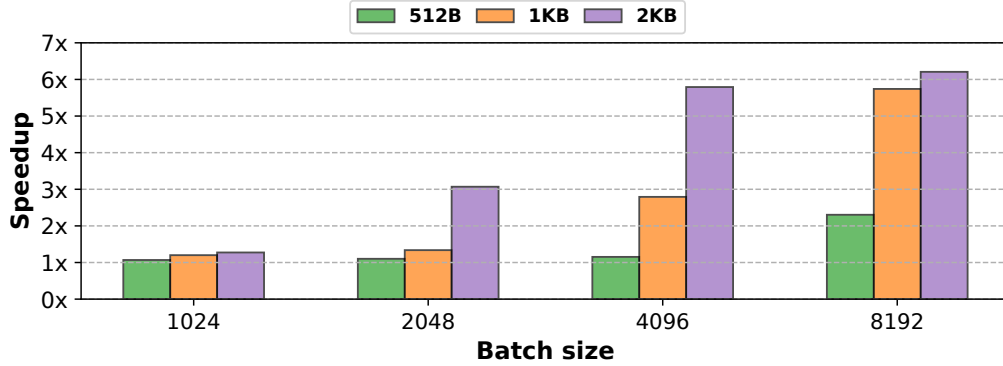


Figure 5.10: Normalized DLRM embedding lookup throughput varying batch and entry sizes.

code. The `CachedEmbeddingBag` maintains the embedding tables in CPU memory, caching frequently accessed embeddings in a buffer in GPU memory. `CachedEmbeddingBag` supports both training and inference. In training, the embedding table values can change, necessitating that evicted table entries from GPU memory have to be written back to the CPU. For inference, these values are static. Hence, we slightly modify `CachedEmbeddingBag` to avoid unnecessary data movement from the GPU back to the CPU. All other settings are default.

Setup. We deployed embedding weights from a pre-trained model from NVIDIA [236] trained on the Criteo 1TB dataset [237], consisting of 26 embedding tables with an embedding entry size of 512B (128×4B floats).

The embedding tables vary greatly in size, with some being as small as 4 entries. We focus on the large tables with at least 25,000 entries. Every lookup involves retrieving an entry from each of these tables. We evaluate the embedding lookup time of `CachedEmbeddingBag` enhanced with IBP compared to the baseline, using different batch sizes. To examine scalability, we rescale the embedding sizes from 512B to 2KB as industry models reach these larger sizes [238].

Performance. Figure 5.10 shows the throughput we obtained. We found that the major source of improvement was due to the GPU-based data transfer that we employ (subsection 5.4.2). In the baseline, the CPU performs the table lookups, which consumes a sig-

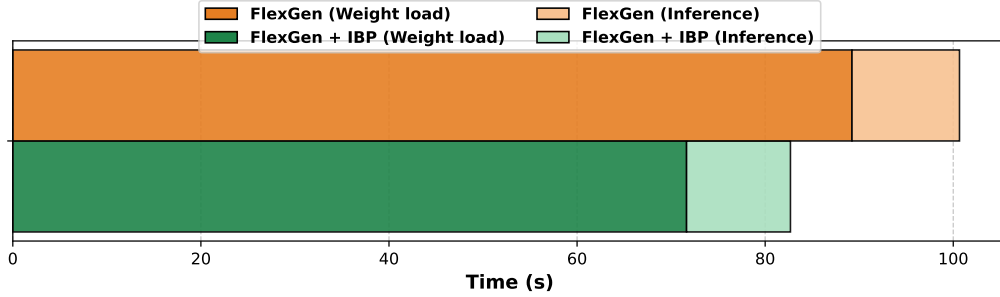


Figure 5.11: LLM inference latency with FlexGen weight offloading.

nificant amount of time. The looked-up entries are then transferred to the GPU. As the batch and entry size increased, the overhead of CPU-based lookups increased. Conversely, IBP provides a single Python function as a drop-in replacement for this operation, which performs the lookup and transfer in parallel on the GPU, aligning the transfers as needed. This avoids the overheads of CPU lookups, using zero-copy to read directly from the GPU. Compression reduced the overhead of transfers by 8% to 20% on average, reducing with increasing batch size.

5.6.4 LLM inference speedup

We test IBP’s impact on LLM KV cache and weight offloading. We enhance two frameworks, FlexGen [23] and InfiniGen [15], with IBP to evaluate weight and KV cache offloading respectively. We use the same workload for both: a query of 1,920 input tokens with a batch size of 20, generating 128 output tokens.

Integrating IBP with FlexGen took ~ 20 lines of Python code for the core logic. The copy function, which transfers weights from the CPU to GPU, is replaced with IBP’s decompress function, and compression is done while loading the weights to CPU memory. We used 64 SMs for decompression, allowing the rest to be used for concurrent inference computation.

We ran Gemma-7B [233], which uses BF16 weights. These are stored in CPU memory, and brought layer-by-layer to the GPU. Figure 5.11 shows the inference latency, breaking down the time spent on weight loading and other inference latencies. We see that IBP makes

the end-to-end latency 22% faster, driven by 24.5% faster weight loading. This improvement directly results from the compression that IBP gets with the BF16 LLM weights.

While the previous experiment demonstrated weight offloading, KV cache transfers from the CPU also incur a heavy overhead. To demonstrate this, we integrate IBP into InfiniGen [15], a recent LLM KV cache management framework tailored for long-context inference that builds on FlexGen [23] minimizing KV cache movement via indexing. This required ~ 40 lines of code changes. It primarily involved replacing the function responsible for fetching the KV cache from CPU memory with our IBP-based implementation. We reproduce an experiment from this work—we use Meta’s OPT model [232], which uses the FP16 data type. We tried the newer Gemma model which uses BF16, but had issues getting it to work with InfiniGen due to conflicts with RoPE [239] operations.

For the KV cache, we do not preprocess data for IBP. Instead, we sample the prompt tokens’ K/V to generate a *Mask* and *Bitval* during each prefill phase. This is used to compress the output token K/V entries during decode. The prefill time is slightly increased as a consequence and the associated overhead is included in our experiment. This increase is compensated by the reduction in transfer time.

Figure 5.12 shows the KV cache transfer and overall inference latencies with and without IBP. IBP reduces KV cache movement latency by up to 37% and reduces overall inference latency by up to 27% versus InfiniGen. IBP preprocessing increases the input processing time by 10%. As input processing accounts for less than 20% of total inference time, the gains in transfer speed outweigh this overhead. The smaller models (2.7B, 6.7B, 13B) see larger inference latency reductions. OPT-30B does not fit in GPU memory and adds overhead for transferring uncompressed model weights, reducing the benefit of IBP’s KV cache compression in the overall inference latency.

The speedup is primarily due to IBP’s GPU-initiated transfer (subsection 5.4.2). InfiniGen relies on the CPU to perform the KV cache indexing and transfer, while IBP uses the GPU. Compression improved transfers by 2%. InfiniGen transfers small 128B – 256B K/V entries per head for OPT, and FP16 does not compress well, limiting the transfer speedup.

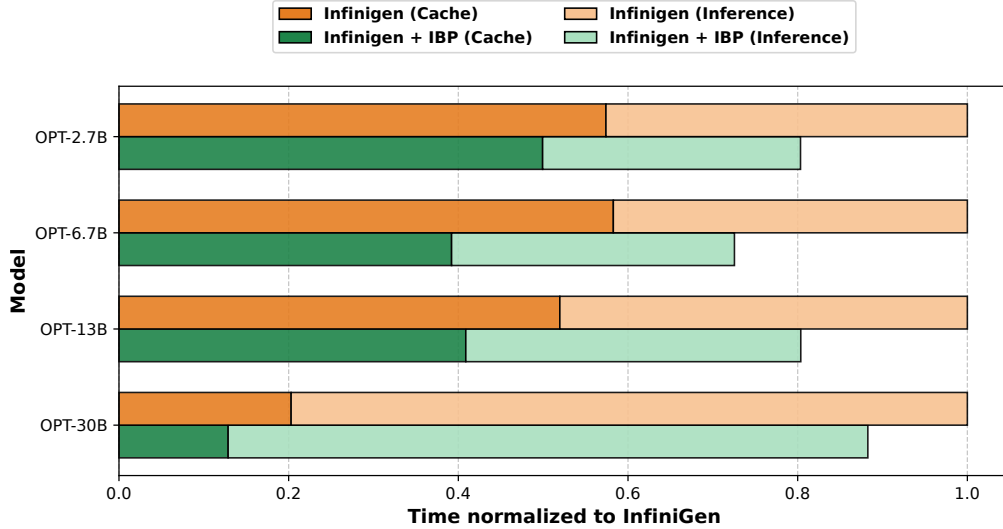


Figure 5.12: Normalized LLM inference latency with InfiniGen KV cache offloading.

These experiments show IBP improves LLM inference through two mechanisms: compression (dominant for BF16 weights) and efficient GPU-initiated transfer (dominant for sparse KV cache).

5.6.5 Performance sensitivity studies

We now examine how performance varies with configuration parameters and design choices. **Chunk Size (CS) and Invariant Threshold (T).** Chunk size and invariant threshold as a fraction of considered tensors (T/N), have an impact on the compression and decompression performance that can be obtained. A warp has 32 threads. To maximize parallelism, at least 32 chunks are desirable. However, each chunk requires a participation bit, affecting the compression ratio. T determines how often a bit needs to have the same value across N tensors to be considered invariant, also affecting the ratio.

In Figure 5.13, we examine the effect of varying CS and T/N on the Products dataset. For 1B CS (400 chunks), the compression ratio is low ($< 3\%$). This is because one participation bit is needed per byte, leading to a 12.5% overhead. As CS increases to 8B (50 chunks), this impact reduces to 1.6%, increasing compression up to 12.7%. For high T/N (e.g., 100%), no

Chunk size	70%	75%	80%	85%	90%	95%	100%
1B	1.8%	2.5%	2.5%	2.5%	2.3%	0.0%	0.0%
2B	7.3%	8.6%	8.6%	8.6%	8.4%	4.8%	0.0%
4B	10.4%	11.7%	11.7%	11.7%	11.5%	7.9%	0.0%
8B	8.7%	12.6%	12.7%	12.7%	12.5%	9.1%	0.0%

Figure 5.13: Space saved with different chunk sizes and invariant threshold fractions T/N for $N = 2.4\text{M}$ tensors in Products.

bits are invariant, while a T/N around 80–85% strikes a good balance. We evaluate varying N in the next subsection.

Clustered Compression (N). So far, we have constructed a single *Mask* and *Bitval* for the entire dataset (setting N to the cardinality of the dataset), exploiting globally invariant bits to provide compression. This works well for the ML datasets we examined, but may be suboptimal for datasets with local trends among data items that are not globally prevalent.

For datasets with local invariance, a smaller N can provide higher compression ratios. We call this *clustered compression*, where the dataset is partitioned into k clusters of size N_c for cluster c , and a *Mask* and *Bitval* is created for each cluster. How the dataset is partitioned into clusters has a major impact on compressibility. A common approach to finding similarity within a dataset is by k -means clustering. This maximizes the chance that local invariance trends among tensors are exploited within each cluster, improving compression. We modify k -means to use tensor bit-similarity for distances instead of absolute values.

We demonstrate this approach in Figure 5.14, which plots the number of clusters k (found via k -means clustering) versus the net space saved (i.e., space savings due to compression minus metadata overhead for additional *Mask* and *Bitval*) across a number of datasets, including datasets of 250,000 tensors containing normally and uniformly distributed random integers. We can see that our GNN dataset (Reddit) resembles a normal distribution, where clustering provides very minor benefit, i.e., the peak was 12.7% savings with 477 clusters,

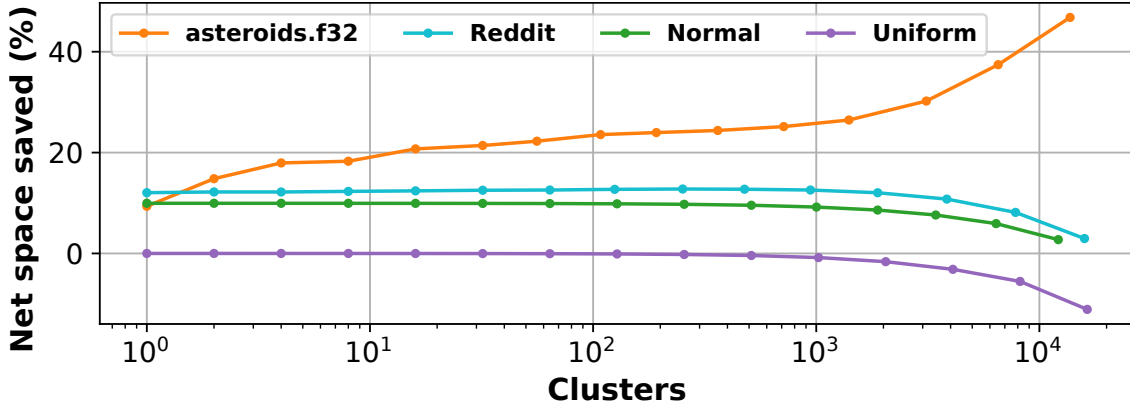


Figure 5.14: Clustered compression net space savings.

compared to 12.0% savings for one cluster. A uniform distribution is difficult to compress, as values have no trends to exploit.

To show that clustering can yield net space savings, we evaluate the *asteroid.f32* dataset [240, 194], containing the pressure component of a 3D simulation of an asteroid impact. The dataset contains a 3D tensor of $500 \times 500 \times 500$ floating point values, which we transformed into a set of 250,000 tensors of size 500. The bit values of this dataset show local invariance trends—as the number of clusters increases, these similar tensors get grouped together and the space savings increase. With a single cluster, we obtain only $\sim 10\%$ net space savings. With 13,725 clusters, the dataset compresses by 58%, yielding a net space saving of 47% ($\sim 11\%$ of the dataset size is consumed by *Mask* and *Bitval* metadata).

Analysis of CPU-based copying. Many existing frameworks rely on the CPU to copy data to/from the GPU, whereas IBP relies on the GPU. We find that the performance of Pytorch’s CPU-based copying varies with the amount of threads dedicated to copying. We previously used 32 threads for our LLM and DLRM baselines. In Figure 5.15 we ran the LLM inference experiment varying the number of CPU threads dedicated towards Infinigen.

We see that as the threads increase, the performance of KV cache management improves, due to more threads performing copying. However, IBP’s GPU-based copying still remains greatly superior. In practical scenarios, dedicating large numbers of threads to copying is of-

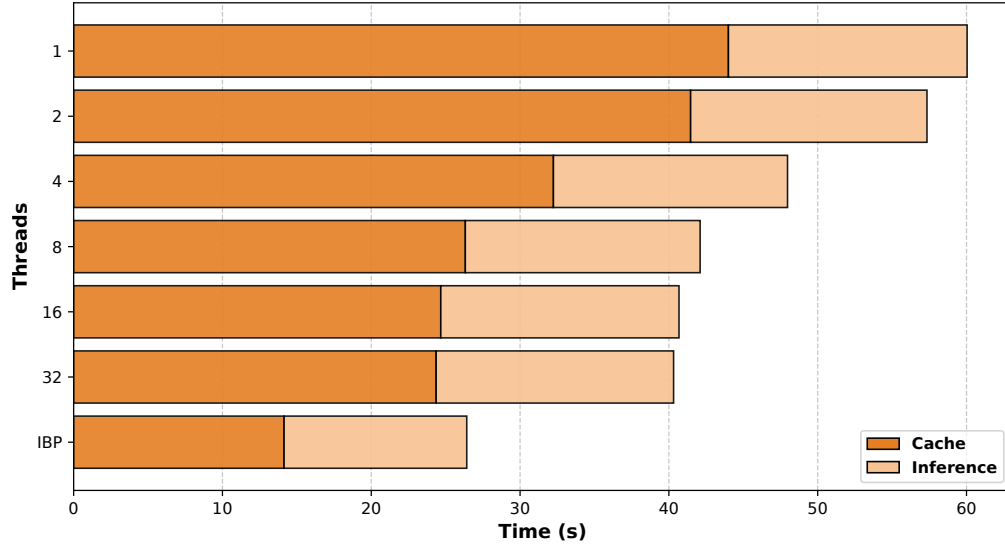


Figure 5.15: LLM inference latency with varying CPU threads.

ten infeasible. The CPU performs many duties during ML inference, like fetching inputs [241] or handling RPC calls [242]. Increasing the number of CPU threads for tasks like copying often moves the bottleneck from PCIe transfers to CPU contention. Conversely, as the GPU is waiting on transfers, making use of its stalled SMs does not increase contention, while alleviating the PCIe transfer bottleneck.

5.7 Conclusion

This chapter discussed IBP, an instantiation of the reducing data movement mitigation technique. As offloading-based ML frameworks rely on moving data from CPU memory during model execution, they are bottlenecked by the PCIe bandwidth. We use lossless compression to alleviate this bottleneck, proposing Invariant Bit Packing (IBP), a novel compression scheme that compresses *across* tensors. Results show the effectiveness of on-the-fly decompression by improving GNN training latency by 74%, DLRM embedding lookups by 180%, and LLM inference by 24% on average. We now look at how these projects inspire future research.

Chapter 6

FUTURE OUTLOOK

This thesis explores three systems that demonstrate different data movement mitigations. Each targets a specific use case: $(MC)^2$ targets copy-heavy workloads that suffer from cache misses, POD-Attention targets hybrid-batch LLM inference, and IBP targets offloaded ML execution. In this chapter we discuss future work inspired by these systems.

6.1 *Coordinated cross-layer copying*

Motivation. Current system proposals delegate copying to a specific device, like DMA-based copying [55], in-memory copying [103], cache-based copying [102], or memory-controller-based copying [11]. Each mechanism has its own set of tradeoffs—DMA engines free the CPU but have high startup cost; in-DRAM copying offers high bandwidth but requires the source and destination to lie within the same module; cache-based copying is fast when data is hot but degrades when it must be fetched from memory. These mechanisms also require using their own set of APIs for copying, to opt-in to the specific system. No mechanism can be uniformly applied across all workloads, and all require application code changes to be used. For optimal performance we need a coordinated approach.

The abstraction for unified copy already exists at the programmer interface: `memcpy` and related functions [97, 96]. What’s missing is the runtime machinery to select the right underlying mechanism for each call. Currently, using a specific mechanism requires explicit programmer effort—DMA engines must be programmed through their own API, while other approaches require using their set of copy instructions. This requires the programmer to first determine the best device to handle copying, then modify code to use that mechanism. This is an infeasible task, as the programmer does not know the system state when the copy is

Table 6.1: Copy coordinator decision table

Data location	Decider	Copy mechanism
Cache	Cache	CPU copy or cache accelerator
DRAM, same module	Memory controller	DRAM accelerator
DRAM, different module	Memory controller	DMA engine

issued, e.g., the state of data in the cache, which impacts which technique is optimal.

Proposal. A coordinated system would route memcpy invocations automatically to the appropriate mechanism as shown in Table 6.1. Cache-resident data would be copied by the CPU or a cache accelerator, taking advantage of the low latency access times of the cache, and avoiding startup overheads present for devices like DMA engines. Cache misses would be forwarded to an in-memory copy unit or DMA engine, depending on the source and destination locations. The programmer’s job becomes simple: call memcpy, then the memory hierarchy handles the rest.

Implementation and challenges. Implementing this requires significant hardware changes. Devices need a dedicated interconnect to communicate with each other and send messages for coordination. However, this is not an impossible task. Cache coherence protocols rely on similar communication between cache layers, distributed across cores. We can extend this to other devices, repurposing it for copy coordination. When coherence indicates that data is available in the cache, the coordination system selects a local copy mechanism, i.e., cache-based copy acceleration or CPU copying. If coherence indicates a cache miss, the message is forwarded to the memory controller, which handles it in an $(MC)^2$ fashion, i.e., the copy is performed lazily and in the background, without CPU involvement. When copying, it chooses between in-DRAM copying and the DMA engine, depending on the DRAM layout.

While the effort is high, the benefit is that copying becomes a ‘solved’ problem—newer copy-offloading schemes only need to be integrated once to benefit all applications.

6.2 Automatic GPU co-scheduling

Motivation. At an industrial scale, GPU utilization has become a critical factor in cost management [243, 244]. GPUs are expensive both to purchase and to operate, with substantial power consumption even when idle. Current methods of GPU sharing—time-slicing or dividing multiprocessors among kernels [245, 246, 247]—multiplex applications but still leave large utilization gaps. Kernels can end up using less than 10% of either compute or bandwidth, leaving significant performance on the table (Chapter 4).

These gaps arise because different workloads are dependent on the GPU for different purposes. Graph workloads [192, 248] rely on large numbers of GPU threads to saturate memory bandwidth, but leave compute largely idle. Matrix multiplication kernels rely on the GPU’s high tensor core throughput, saturating compute while leaving memory bandwidth idle. Both show the GPU as completely busy in standard utilization metrics, yet each uses only a fraction of the GPU’s total resources.

Proposal. We saw in Chapter 4 that co-locating operations with complementary resource profiles can improve a largely overlooked metric—intra-GPU utilization. This opens a new dimension of multiplexing: rather than dividing multiprocessors among kernels, divide the compute and memory resources within each multiprocessor across complementary kernels. Memory-bound kernels co-located with compute-bound kernels can saturate both resource classes simultaneously, with each getting the resource it primarily needs, avoiding slowdown.

Implementation and challenges. A source-to-source compiler tool could fuse complementary pairs to maximize utilization while minimizing interference. Fusion requires tuning, as we require the resource usage of each component kernel to be similar so that the fused kernel preserves this usage. For example, if a kernel requires a large number of registers, while another does not, the fused kernel would allocate the maximum of the two, wasting registers on the smaller component kernel. Managing this usage isn’t impossible, however, as register usage is known during compilation and can be adjusted through compiler flags. To determine resource utilization, we must profile the kernels offline and note their usage.

Based on this profiling, we generate candidate kernel pairs then perform fusion.

The simplest case is fusing parallel kernels within a single application, where trust is already established and throughput is the main metric. Fusing across applications provides more opportunities for cost savings but raises real concerns: side-channel attacks, isolation breakdown, and information leaks. Existing GPU isolation primitives (MIG, MPS) provide a foundation, but extending them to fine-grained intra-SM co-location is an open problem. In addition, offline profiling may not capture kernel variations during deployment, requiring online tuning as well. Ultimately, hardware support is required to provide better control over multiprocessor scheduling, resource isolation, and more fine-grained utilization metrics.

6.3 Pervasive lossless compression

Motivation. IBP showed that lossless compression can be practical for ML workloads—but it is just one algorithm among many. Different data representations, access patterns, and memory/interconnect technologies benefit from compression strategies in different ways. Data movement occurs across a wide variety of interconnects: PCIe between CPU and peripheral devices, NVLink between GPUs, and InfiniBand/Ethernet for distributed deployments. Each is bandwidth-limited and a potential bottleneck.

Compression is often retrofitted into systems after they are built. The barrier is deployment complexity: many algorithms exist, data responds differently, and selecting and integrating the right one for each data path is non-trivial. This deters early integration. As workloads scale and bandwidth continues to lag compute, pervasive lossless compression across the stack becomes increasingly important.

Proposal. A practical solution is a runtime compression module that examines data characteristics and selects an appropriate algorithm for each transfer path. Algorithms like IBP, Cascaded, and zStd have different strengths; a module that picks among them based on the data and target memory/interconnect would let applications benefit from compression without requiring application designers to manage it manually.

The module should take into consideration the specific characteristics of the intercon-

nects that data is expected to traverse. For each machine it is deployed on, the module would measure the performance of different algorithms across each link (e.g., memory, PCIe, NVLink). For low-bandwidth links, high compressibility is important, as the volume of data reduced translates to improvement. For high-bandwidth links, decompression throughput is the key requirement, as the act of decompressing shouldn't negate transfer improvements.

Implementation and challenges. When a transfer is requested during runtime, the module would sample a subset of data to examine its compressibility with different algorithms. If the format of the data is not expected to change in the future, then the same algorithm is reused. For example, in distributed LLM inference, there are frequent collective communication patterns [210]. The data involved is similar, and so an algorithm identified as efficient for the first communication can be reused for further iterations. Thus, this preprocessing is done once and amortized across iterations.

As a further extension, we have each link maintain its own compression schemes in hardware, completely hiding the implementation from the programmer. Some vendors have already begun doing this—newer NVIDIA GPUs offer hardware support for decompression [249]. This generalizes compression, making it easier to integrate into a wider variety of applications.

Chapter 7

CONCLUSION

7.1 *Concluding remarks*

We examined the growing gap between compute and data movement throughput in modern systems, and developed mitigations for memory-intensive workloads that minimize deployment effort. As compute outpaces memory and interconnect bandwidth, data movement becomes the dominant bottleneck for a diverse set of workloads including databases, LLMs, DLRMs, and GNNs.

We argue that effectively reducing this bottleneck requires identifying key application characteristics that enable mitigations. We classify mitigations into three categories: (1) avoiding data movement that applications do not necessarily need, (2) hiding data movement by overlapping it with complementary work, and (3) reducing the volume of data transferred by exploiting patterns within the data. Each category identifies an application property and prescribes how to exploit it, providing a composable framework for analyzing data movement bottlenecks across diverse workloads.

We demonstrate the framework through three systems. $(MC)^2$ extends the memory controller to lazily track copies, avoiding data movement for copies that are never accessed. POD-Attention fuses the prefill and decode phases of LLM inference, hiding decode’s memory stalls behind prefill’s compute. IBP losslessly compresses ML tensors, exploiting low-entropy, invariant bits present across tensors, reducing PCIe data transfers. $(MC)^2$ achieves up to 78% higher throughput for database transactions and $250\times$ lower latency on huge-page copy-on-write faults. POD-Attention yields up to 59% faster LLM attention. IBP delivers on average 74% faster GNN training, 180% faster DLRM embedding lookups, and 24% faster offloaded LLM inference.

7.2 *Broader implications*

Fully eliminating data movement overheads requires restructuring the hardware-software stack itself. Processing in memory (PIM) and near-memory compute (NMC) [44, 45] are architecture designs that move compute to where data lives, rather than moving data to remote compute units. DRAM has higher internal bandwidth than the on-chip network, making it efficient to dispatch computation to the data. Dataflow architectures [250, 251, 252] take a related approach, organizing execution around data dependencies rather than control flow. These approaches can substantially reduce the data movement problem, but their adoption requires new hardware deployments and significant application rewriting.

The framework presented in this thesis is complementary: as new layers of the memory hierarchy emerge—like advancements in CXL, NVLink, and HBM—avoiding, hiding, and reducing data movement remain a useful method for identifying mitigation opportunities on commodity hardware.

BIBLIOGRAPHY

- [1] NVIDIA, “Nvidia v100 tensor core gpu,” <https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf>.
- [2] —, “Nvidia a100 tensor core gpu,” <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf>.
- [3] —, “Nvidia h100 tensor core gpu,” <https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet>.
- [4] —, “Nvidia blackwell b200 datasheet,” <https://www.primeline-solutions.com/media/categories/server/nach-gpu/nvidia-hgx-h200/nvidia-blackwell-b200-datasheet.pdf>.
- [5] Anthropic, “Anthropic claude,” <https://claude.ai>.
- [6] OpenAI, “Chatgpt,” <https://chat.openai.com>.
- [7] Y. Chronis, A. Ailamaki, L. Benson, H. Caminal, J. Gičeva, D. Patterson, E. Seldar, and L. W. Wills, “Databases in the era of memory-centric computing,” in *Proceedings of the Conference on Innovative Data Systems Research*, 2025.
- [8] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, “Ai and memory wall,” *IEEE Micro*, vol. 44, no. 3, pp. 33–39, 2024.
- [9] D. R. K. Ports and K. Grittner, “Serializable snapshot isolation in postgresql,” *Proc. VLDB Endow.*, vol. 5, no. 12, p. 1850–1861, Aug. 2012. [Online]. Available: <https://doi.org/10.14778/2367502.2367523>
- [10] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, “An empirical evaluation of in-memory multi-version concurrency control,” *Proc. VLDB Endow.*, vol. 10, no. 7, p. 781–792, mar 2017. [Online]. Available: <https://doi.org/10.14778/3067421.3067427>
- [11] A. K. Kamath and S. Peter, “(mc)²: Lazy memcopy at the memory controller,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. Los Alamitos, CA, USA: IEEE Computer Society, Jul. 2024, pp. 1112–1128. [Online]. Available: <https://doi.org/10.1109/ISCA59077.2024.00084>

- [12] 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*, ser. ASPLOS 2025. New York, NY, USA: Association for Computing Machinery, 2025, p. 897–912. [Online]. Available: <https://doi.org/10.1145/3676641.3715996>
- [13] R. Sanovar, S. Bharadwaj, R. S. Amant, V. Rühle, and S. Rajmohan, “Leanattention: Hardware-aware scalable attention mechanism for the decode-phase of transformers,” in *Eighth Conference on Machine Learning and Systems*, 2025. [Online]. Available: <https://openreview.net/forum?id=KVZDNEoC0Q>
- [14] W. Wu, X. Shi, L. He, and H. Jin, “Turbognn: Improving the end-to-end performance for sampling-based gnn training on gpus,” *IEEE Transactions on Computers*, vol. 72, no. 9, pp. 2571–2584, 2023.
- [15] W. Lee, J. Lee, J. Seo, and J. Sim, “InfiniGen: Efficient generative inference of large language models with dynamic KV cache management,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 155–172. [Online]. Available: <https://www.usenix.org/conference/osdi24/presentation/lee>
- [16] J. Sun, L. Su, Z. Shi, W. Shen, Z. Wang, L. Wang, J. Zhang, Y. Li, W. Yu, J. Zhou, and F. Wu, “Legion: Automatically pushing the envelope of Multi-GPU system for Billion-Scale GNN training,” in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 165–179. [Online]. Available: <https://www.usenix.org/conference/atc23/presentation/sun>
- [17] A. K. Kamath, A. Krishnamurthy, M. Canini, and S. Peter, “Reducing the gpu memory bottleneck with lossless compression for ml,” in *Proceedings of the 21st European Conference on Computer Systems*, ser. EUROSYS ’26. New York, NY, USA: Association for Computing Machinery, 2026, p. 899–918. [Online]. Available: <https://doi.org/10.1145/3767295.3803595>
- [18] H. Zhao, Z. Han, Z. Yang, Q. Zhang, M. Li, F. Yang, Q. Zhang, B. Li, Y. Yang, L. Qiu, L. Zhang, and L. Zhou, “Silod: A co-design of caching and scheduling for deep learning clusters,” in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 883–898. [Online]. Available: <https://doi.org/10.1145/3552326.3567499>

- [19] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, “Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills,” 2023. [Online]. Available: <https://arxiv.org/abs/2308.16369>
- [20] X. Zhang, Y. Shen, Y. Shao, and L. Chen, “Ducati: A dual-cache training system for graph neural networks on giant graphs with the gpu,” *Proc. ACM Manag. Data*, vol. 1, no. 2, Jun. 2023. [Online]. Available: <https://doi.org/10.1145/3589311>
- [21] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo, “BGL: GPU-Efficient GNN training by optimizing graph data I/O and preprocessing,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 103–118. [Online]. Available: <https://www.usenix.org/conference/nsdi23/presentation/liu-tianfeng>
- [22] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood, “Understanding training efficiency of deep learning recommendation models at scale,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 802–814.
- [23] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Re, I. Stoica, and C. Zhang, “FlexGen: High-throughput generative inference of large language models with a single GPU,” in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 23–29 Jul 2023, pp. 31 094–31 116. [Online]. Available: <https://proceedings.mlr.press/v202/sheng23a.html>
- [24] A. Dutta, S. Krishnan, N. Kwatra, and R. Ramjee, “Accuracy is not all you need,” in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 124 347–124 390. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/file/e0e956681b04ac126679e8c7dd706b2e-Paper-Conference.pdf
- [25] T. Dao, D. Haziza, F. Massa, and G. Sizov, “Flash-Decoding for long-context inference,” <https://crfm.stanford.edu/2023/10/12/flashdecoding.html>, 2023.
- [26] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, and L. Ceze, “Flashinfer: Efficient and customizable attention engine for llm inference serving,” 2025. [Online]. Available: <https://arxiv.org/abs/2501.01005>

- [27] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, “SGLang: Efficient execution of structured language model programs,” in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: <https://openreview.net/forum?id=VqkAKQibpq>
- [28] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture,” *Commun. ACM*, vol. 62, no. 2, p. 48–60, Jan. 2019. [Online]. Available: <https://doi.org/10.1145/3282307>
- [29] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, “Green ai,” *Commun. ACM*, vol. 63, no. 12, p. 54–63, Nov. 2020. [Online]. Available: <https://doi.org/10.1145/3381831>
- [30] 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*, vol. 9, no. 5, pp. 256–268, 1974.
- [31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
- [32] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–12. [Online]. Available: <https://doi.org/10.1145/3079856.3080246>
- [33] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, “Gpu computing,” *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

- [34] W. A. Wulf and S. A. McKee, “Hitting the memory wall: implications of the obvious,” *SIGARCH Comput. Archit. News*, vol. 23, no. 1, p. 20–24, Mar. 1995. [Online]. Available: <https://doi.org/10.1145/216585.216588>
- [35] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé, “Zerializer: Towards zero-copy serialization,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 206–212. [Online]. Available: <https://doi.org/10.1145/3458336.3465283>
- [36] D. Raghavan, S. Ravi, G. Yuan, P. Thaker, S. Srivastava, M. Murray, P. H. Penna, A. Ousterhout, P. Levis, M. Zaharia, and I. Zhang, “Cornflakes: Zero-copy serialization for microsecond-scale networking,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 200–215. [Online]. Available: <https://doi.org/10.1145/3600006.3613137>
- [37] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, “Arrakis: The operating system is the control plane,” *ACM Trans. Comput. Syst.*, vol. 33, no. 4, Nov. 2015. [Online]. Available: <https://doi.org/10.1145/2812806>
- [38] T. Stamler, D. Hwang, A. Raybuck, W. Zhang, and S. Peter, “zIO: Accelerating IO-Intensive applications with transparent Zero-Copy IO,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 431–445. [Online]. Available: <https://www.usenix.org/conference/osdi22/presentation/stamler>
- [39] J. M. Smith and G. Q. Maguire Jr, “Effects of copy-on-write memory management on the response time of unix fork operations,” *Computing Systems*, vol. 1, no. 3, pp. 255–278, 1988.
- [40] A. J. Smith, “Cache memories,” *ACM Comput. Surv.*, vol. 14, no. 3, p. 473–530, Sep. 1982. [Online]. Available: <https://doi.org/10.1145/356887.356892>
- [41] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA ’90. New York, NY, USA: Association for Computing Machinery, 1990, p. 364–373. [Online]. Available: <https://doi.org/10.1145/325164.325162>

- [42] X. Song, Y. Zhang, R. Chen, and H. Chen, “Ugache: A unified gpu cache for embedding-based deep learning,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 627–641. [Online]. Available: <https://doi.org/10.1145/3600006.3613169>
- [43] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, “Scaling memcache at facebook,” in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 385–398. [Online]. Available: <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala>
- [44] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, “Enabling practical processing in and near memory for data-intensive computing,” in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC ’19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: <https://doi.org/10.1145/3316781.3323476>
- [45] 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*, ser. MICRO ’52. New York, NY, USA: Association for Computing Machinery, 2019, p. 100–113. [Online]. Available: <https://doi.org/10.1145/3352460.3358260>
- [46] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 105–117. [Online]. Available: <https://doi.org/10.1145/2749469.2750386>
- [47] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, “Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA ’16. IEEE Press, 2016, p. 27–39. [Online]. Available: <https://doi.org/10.1109/ISCA.2016.13>
- [48] P. Michaud, “Best-offset hardware prefetching,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480.
- [49] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, “Perceptron-based prefetch filtering,” in *Proceedings of the 46th International*

- Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–13. [Online]. Available: <https://doi.org/10.1145/3307650.3322207>
- [50] S. Pakalapati and B. Panda, “Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 118–131.
- [51] A. Jain, H. Lin, C. Villavieja, B. Kasikci, C. Kennelly, M. Hashemi, and P. Ranganathan, “Limoncello: Prefetchers for scale,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 577–590. [Online]. Available: <https://doi.org/10.1145/3620666.3651373>
- [52] S. Kumar, D. E. Culler, and R. A. Popa, “MAGE: Nearly zero-cost virtual memory for secure computation,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 367–385. [Online]. Available: <https://www.usenix.org/conference/osdi21/presentation/kumar>
- [53] S. Ainsworth and T. M. Jones, “Software prefetching for indirect memory accesses,” in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 305–317.
- [54] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz, “Apt-get: profile-guided timely software prefetching,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 747–764. [Online]. Available: <https://doi.org/10.1145/3492321.3519583>
- [55] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda, “Designing efficient asynchronous memory operations using hardware copy engine: A case study with i/oat,” in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8.
- [56] J. He, Y. Dong, D. Du, M. Zou, Z. Yu, Y. Ren, N. Jia, Y. Xia, and H. Chen, “How to copy memory? coordinated asynchronous copy as a first-class os service,” in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, ser. SOSP '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1062–1081. [Online]. Available: <https://doi.org/10.1145/3731569.3764800>

- [57] W. Su, L. Wang, M. Su, and S. Liu, “A processor-dma-based memory copy hardware accelerator,” in *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*, 2011, pp. 225–229.
- [58] K. Zhu, Y. Zhao, L. Zhao, G. Zuo, Y. Gu, D. Xie, Y. Gao, Q. Xu, T. Tang, Z. Ye, K. Kamahori, C.-Y. Lin, S. Wang, A. Krishnamurthy, and B. Kasikci, “Nanoflow: Towards optimal large language model serving throughput,” 2024. [Online]. Available: <https://arxiv.org/abs/2408.12757>
- [59] S. Wang, J. Wei, A. Sabne, A. Davis, B. Ilbeyi, B. Hechtman, D. Chen, K. S. Murthy, M. Maggioni, Q. Zhang, S. Kumar, T. Guo, Y. Xu, and Z. Zhou, “Overlap communication with dependent computation via decomposition in large deep learning models,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2022, p. 93–106. [Online]. Available: <https://doi.org/10.1145/3567955.3567959>
- [60] S. Rashidi, M. Denton, S. Sridharan, S. Srinivasan, A. Suresh, J. Nie, and T. Krishna, “Enabling compute-communication overlap in distributed deep learning training platforms,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 540–553.
- [61] R. Gond, N. Kwatra, and R. Ramjee, “Tokenweave: Efficient compute-communication overlap for distributed llm inference,” 2026. [Online]. Available: <https://arxiv.org/abs/2505.11329>
- [62] B. Widrow, I. Kollar, and M.-C. Liu, “Statistical theory of quantization,” *IEEE Transactions on Instrumentation and Measurement*, vol. 45, no. 2, pp. 353–361, 1996.
- [63] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, “Smoothquant: Accurate and efficient post-training quantization for large language models,” 2023.
- [64] Y. Zhao, C.-Y. Lin, K. Zhu, Z. Ye, L. Chen, S. Zheng, L. Ceze, A. Krishnamurthy, T. Chen, and B. Kasikci, “Atom: Low-bit quantization for efficient and accurate llm serving,” in *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. D. Sa, Eds., vol. 6, 2024, pp. 196–209. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2024/file/5edb57c05c81d04beb716ef1d542fe9e-Paper-Conference.pdf
- [65] E. Frantar, S. Ashkboos, T. Hoeffler, and D. Alistarh, “Gptq: Accurate post-training quantization for generative pre-trained transformers,” 2023.

- [66] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, “Gpt3.int8(): 8-bit matrix multiplication for transformers at scale,” in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 30 318–30 332. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/c3ba4962c05c49636d4c6206a97e9c8a-Paper-Conference.pdf
- [67] M. Zhu and S. Gupta, “To prune, or not to prune: exploring the efficacy of pruning for model compression,” 2017. [Online]. Available: <https://arxiv.org/abs/1710.01878>
- [68] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, “Exploring sparsity in recurrent neural networks,” 2017. [Online]. Available: <https://arxiv.org/abs/1704.05119>
- [69] M. Hershcovitch, A. Wood, L. Choshen, G. Girmonsky, R. Leibovitz, O. Ozeri, I. Ennmouri, M. Malka, P. Chin, S. Sundararaman, and D. Harnik, “Zipnn: Lossless compression for ai models,” in *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, 2025, pp. 186–198.
- [70] N. Sakharlykh, D. LaSalle, and B. Karsin, “Optimizing data transfer using lossless compression with nvidia nvcomp,” <https://developer.nvidia.com/blog/optimizing-data-transfer-using-lossless-compression-with-nvcomp/>, 2020.
- [71] D. R. Carvalho and A. Sez nec, “Understanding cache compression,” *ACM Trans. Archit. Code Optim.*, vol. 18, no. 3, Jun. 2021. [Online]. Available: <https://doi.org/10.1145/3457207>
- [72] R. L. Sites, “Fast memcpy, a system design,” <https://www.sigarch.org/fast-memcpy-a-system-design/>, Dec 2022.
- [73] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, “Profiling a warehouse-scale computer,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 158–169. [Online]. Available: <https://doi.org/10.1145/2749469.2750392>
- [74] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, “Attack of the killer microseconds,” *Commun. ACM*, vol. 60, no. 4, p. 48–54, mar 2017. [Online]. Available: <https://doi.org/10.1145/3015146>
- [75] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, “Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization,”

- in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, ser. SIGMETRICS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 323–336. [Online]. Available: <https://doi.org/10.1145/2896377.2901453>
- [76] Crucial, “Ddr5 memory: Everything you need to know,” <https://www.crucial.in/articles/about-memory/everything-about-ddr5-ram>, 2023.
- [77] 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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 574–587. [Online]. Available: <https://doi.org/10.1145/3575693.3578835>
- [78] P. Duraisamy, W. Xu, S. Hare, R. Rajwar, D. Culler, Z. Xu, J. Fan, C. Kennelly, B. McCloskey, D. Mijailovic, B. Morris, C. Mukherjee, J. Ren, G. Thelen, P. Turner, C. Villavieja, P. Ranganathan, and A. Vahdat, “Towards an adaptable systems architecture for memory tiering at warehouse-scale,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 727–741. [Online]. Available: <https://doi.org/10.1145/3582016.3582031>
- [79] C. E. Link, “Compute express link™: The breakthrough cpu-to-device interconnect cxl,” <https://www.computeexpresslink.org/>.
- [80] Intel, “Intel® optane™ persistent memory,” <https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html>.
- [81] Google, “Protocol buffers,” <https://protobuf.dev/>, October 2023.
- [82] D. Zheng, X. Song, C. Yang, D. LaSalle, and G. Karypis, “Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs,” in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 4582–4591. [Online]. Available: <https://doi.org/10.1145/3534678.3539177>
- [83] T. Kaler, N. Stathas, A. Ouyang, A.-S. Iliopoulos, T. Schardl, C. E. Leiserson, and J. Chen, “Accelerating training and inference of graph neural networks with fast sampling and pipelining,” in *Proceedings of Machine Learning and*

- Systems*, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, 2022, pp. 172–189. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2022/file/afacc5db3e0e85b446e6c7727cd7dca5-Paper.pdf
- [84] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, “A hardware accelerator for protocol buffers,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 462–478. [Online]. Available: <https://doi.org/10.1145/3466752.3480051>
- [85] A. C. De Melo, “The new linux’perf’tools,” in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.
- [86] Google, “Fleetbench,” <https://github.com/google/fleetbench>, 2023.
- [87] MongoDB, “Mongodb,” github.com/mongodb/mongo/blob/v4.4/.
- [88] H. Lim, M. Kaminsky, and D. G. Andersen, “Cicada: Dependably fast multi-core in-memory transactions,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 21–35. [Online]. Available: <https://doi.org/10.1145/3035918.3064015>
- [89] A. Kemper and T. Neumann, “Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots,” in *2011 IEEE 27th International Conference on Data Engineering*, 2011, pp. 195–206.
- [90] L. Li, G. Wang, G. Wu, Y. Yuan, L. Chen, and X. Lian, “A comparative study of consistent snapshot algorithms for main-memory database systems,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 2, pp. 316–330, 2021.
- [91] Redis, “Optimizing redis: Diagnosing latency issues,” <https://redis.io/docs/management/optimization/latency/>.
- [92] P. Hudak, “Conception, evolution, and application of functional programming languages,” *ACM Comput. Surv.*, vol. 21, no. 3, p. 359–411, sep 1989. [Online]. Available: <https://doi.org/10.1145/72551.72554>
- [93] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, “Janus: Optimizing memory and storage support for non-volatile memory systems,” in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 143–156. [Online]. Available: <https://doi.org/10.1145/3307650.3322206>

- [94] J. Zhang, M. Swift, and J. J. Li, “Software-defined address mapping: A case on 3d memory,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 70–83. [Online]. Available: <https://doi.org/10.1145/3503222.3507774>
- [95] S. Yadalam, N. Shah, X. Yu, and M. Swift, “Asap: A speculative approach to persistence,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 892–907.
- [96] G. Chatelet, C. Kennelly, S. L. Xi, O. Sykora, C. Courbet, X. D. Li, and B. De Backer, “Automemcpy: A framework for automatic generation of fundamental memory operations,” in *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 39–51. [Online]. Available: <https://doi.org/10.1145/3459898.3463904>
- [97] G. Kappes and S. V. Anastasiadis, “Asterope: A cross-platform optimization method for fast memory copy,” in *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, ser. PLOS ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 9–16. [Online]. Available: <https://doi.org/10.1145/3477113.3487264>
- [98] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer, “Architecture support for improving bulk memory copying and initialization performance,” in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009, pp. 169–180.
- [99] S. Wong, F. Duarte, and S. Vassiliadis, “A hardware cache memcpy accelerator,” in *2006 IEEE International Conference on Field Programmable Technology*, 2006, pp. 141–148.
- [100] F. Duarte and S. Wong, “A memcpy hardware accelerator solution for non cache-line aligned copies,” in *2007 IEEE International Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, 2007, pp. 397–402.
- [101] S. Vassiliadis, F. Duarte, and S. Wong, “A load/store unit for a memcpy hardware accelerator,” in *2007 International Conference on Field Programmable Logic and Applications*, 2007, pp. 537–541.
- [102] F. Duarte and S. Wong, “Cache-based memory copy hardware accelerator for multicore systems,” *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1494–1507, 2010.

- [103] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, “Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 568–580.
- [104] 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 *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 185–197. [Online]. Available: <https://doi.org/10.1145/2540708.2540725>
- [105] H. Mao, R. H. Katz, and K. Asanović, “Hardware acceleration for memory to memory copies,” *Master’s thesis*, 2017.
- [106] Z. Chen, D. Li, Z. Wang, H. Liu, and Y. Tang, “Ramci: a novel asynchronous memory copying mechanism based on i/oat,” *CCF Transactions on High Performance Computing*, vol. 3, pp. 129–143, 2021.
- [107] L. Zhao, L. N. Bhuyan, R. Iyer, S. Makineni, and D. Newell, “Hardware support for accelerating data movement in server platform,” *IEEE Transactions on Computers*, vol. 56, no. 6, pp. 740–753, 2007.
- [108] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam, “The demikernel datapath os architecture for microsecond-scale datacenter systems,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 195–211. [Online]. Available: <https://doi.org/10.1145/3477132.3483569>
- [109] D. Raghavan, P. Levis, M. Zaharia, and I. Zhang, “Breakfast of champions: Towards zero-copy serialization with nic scatter-gather,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 199–205. [Online]. Available: <https://doi.org/10.1145/3458336.3465287>
- [110] A. M. Devices, “Amd64 architecture programmer’s manual volume 2: System programming,” *AMD64 Architecture Programmer’s Manual*, 2024.
- [111] H. Ye, “Introduction to 5-level paging in 3rd gen intel xeon scalable processors with linux,” *Lenovo Press*, 2021.

- [112] K. Zhao, K. Xue, Z. Wang, D. Schatzberg, L. Yang, A. Manousis, J. Weiner, R. Van Riel, B. Sharma, C. Tang, and D. Skarlatos, “Contiguitas: The pursuit of physical memory contiguity in datacenters,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3579371.3589079>
- [113] Intel, “Intel® 64 and ia-32 architectures software developer’s manual,” 2011.
- [114] S. Owens, S. Sarkar, and P. Sewell, “A better x86 memory model: X86-tso,” in *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, ser. TPHOLs '09. Berlin, Heidelberg: Springer-Verlag, 2009, p. 391–407. [Online]. Available: https://doi.org/10.1007/978-3-642-03359-9_27
- [115] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: <https://doi.org/10.1145/2024716.2024718>
- [116] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, “Cacti 7: New tools for interconnect exploration in innovative off-chip memories,” *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, jun 2017. [Online]. Available: <https://doi.org/10.1145/3085572>
- [117] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony, “2.2 amd chiplet architecture for high-performance server and desktop products,” in *2020 IEEE International Solid-State Circuits Conference - (ISSCC)*, 2020, pp. 44–45.
- [118] L. Foundation, “userfaultfd(2),” <http://man7.org/linux/man-pages/man2/userfaultfd.2.html>, February 2020.
- [119] T. Stamler, D. Hwang, A. Raybuck, W. Zhang, and S. Peter, “zio’s copy interposer for mongodb,” https://github.com/tstamler/zIO/blob/mongo/tas/lib/sockets/copy_interpose.c, 2022.
- [120] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *2010 ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: <https://doi.org/10.1145/1807128.1807152>

- [121] S. T. Leutenegger and D. Dias, “A modeling study of the tpc-c benchmark,” in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’93. New York, NY, USA: Association for Computing Machinery, 1993, p. 22–31. [Online]. Available: <https://doi.org/10.1145/170035.170042>
- [122] K. Zhao, S. Gong, and P. Fonseca, “On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications,” in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 540–555. [Online]. Available: <https://doi.org/10.1145/3447786.3456258>
- [123] KeyDB, “Troubleshooting latency issues,” <https://docs.keydb.dev/docs/latency/>.
- [124] L. Zhao, R. Iyer, S. Makineni, L. Bhuyan, and D. Newell, “Hardware support for bulk data movement in server platforms,” in *2005 International Conference on Computer Design*, 2005, pp. 53–60.
- [125] “AI Infrastructure Spending Forecast to Be Over a Trillion Dollars Over the Next Five Years,” <https://www.delloro.com/news/ai-infrastructure-spending-forecast-to-be-over-a-trillion-dollars-over-the-next-five-years/>, 2024.
- [126] “The State of AI Infrastructure at Scale 2024,” <https://ai-infrastructure.org/wp-content/uploads/2024/03/The-State-of-AI-Infrastructure-at-Scale-2024.pdf>, 2024.
- [127] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, “Split-wise: Efficient generative llm inference using phase splitting,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 118–132.
- [128] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, “Orca: A distributed serving system for Transformer-Based generative models,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 521–538. [Online]. Available: <https://www.usenix.org/conference/osdi22/presentation/yu>
- [129] 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*, ser. SOSP ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 611–626. [Online]. Available: <https://doi.org/10.1145/3600006.3613165>

- [130] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, “Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 117–134. [Online]. Available: <https://www.usenix.org/conference/osdi24/presentation/agrawal>
- [131] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, “DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 193–210. [Online]. Available: <https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin>
- [132] B. Wu, S. Liu, Y. Zhong, P. Sun, X. Liu, and X. Jin, “Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism,” in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, ser. SOSP ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 640–654. [Online]. Available: <https://doi.org/10.1145/3694715.3695948>
- [133] A. Agrawal, J. Chen, Íñigo Goiri, R. Ramjee, C. Zhang, A. Tumanov, and E. Choukse, “Mnemosyne: Parallelization strategies for efficiently serving multi-million context length llm inference requests without approximations,” 2024. [Online]. Available: <https://arxiv.org/abs/2409.17264>
- [134] A. Agrawal, N. Kedia, J. Mohan, A. Panwar, N. Kwatra, B. S. Gulavani, R. Ramjee, and A. Tumanov, “Vidur: A large-scale simulation framework for llm inference,” *Proceedings of The Seventh Annual Conference on Machine Learning and Systems, 2024, Santa Clara*, 2024.
- [135] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, “Flashattention: fast and memory-efficient exact attention with io-awareness,” in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS ’22. Red Hook, NY, USA: Curran Associates Inc., 2022.
- [136] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

- [137] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, Y. Dong, Y. Wang *et al.*, “Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics,” *Proceedings of Machine Learning and Systems*, vol. 6, pp. 148–161, 2024.
- [138] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin, A. Bakhtiari, L. Kurilenko, and Y. He, “Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference,” 2024. [Online]. Available: <https://arxiv.org/abs/2401.08671>
- [139] T. Dao, “Flashattention-2: Faster attention with better parallelism and work partitioning,” in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=mZn2Xyh9Ec>
- [140] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, and A. A.-D. et al., “The llama 3 herd of models,” 2024. [Online]. Available: <https://arxiv.org/abs/2407.21783>
- [141] . AI, :, A. Young, B. Chen, C. Li, C. Huang, G. Zhang, G. Zhang, H. Li, J. Zhu, J. Chen, J. Chang, K. Yu, P. Liu, Q. Liu, S. Yue, S. Yang, S. Yang, T. Yu, W. Xie, W. Huang, X. Hu, X. Ren, X. Niu, P. Nie, Y. Xu, Y. Liu, Y. Wang, Y. Cai, Z. Gu, Z. Liu, and Z. Dai, “Yi: Open foundation models by 01.ai,” 2024.
- [142] “CUDA C Programming Guide – Hardware Implementation,” <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#hardware-implementation>, 2024.
- [143] “Parallel Thread Execution ISA Version 8.5 – Cooperative Thread Arrays,” <https://docs.nvidia.com/cuda/parallel-thread-execution/#cooperative-thread-arrays>, 2024.
- [144] “Sarathi-Serve,” <https://github.com/microsoft/sarathi-serve>, 2024.
- [145] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, “Flashattention-3: Fast and accurate attention with asynchrony and low-precision,” 2024. [Online]. Available: <https://arxiv.org/abs/2407.08608>
- [146] M. Bauer, H. Cook, and B. Khailany, “Cudadma: optimizing gpu memory bandwidth via warp specialization,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: <https://doi.org/10.1145/2063384.2063400>
- [147] M. Osama, D. Merrill, C. Cecka, M. Garland, and J. D. Owens, “Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on

- the gpu,” in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 429–431. [Online]. Available: <https://doi.org/10.1145/3572848.3577479>
- [148] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, “Improving gpgpu concurrency with elastic kernels,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 407–418. [Online]. Available: <https://doi.org/10.1145/2451116.2451160>
- [149] H. Zhao, W. Cui, Q. Chen, and M. Guo, “Ispa: Exploiting intra-sm parallelism in gpus via fine-grained resource management,” *IEEE Transactions on Computers*, vol. 72, no. 5, pp. 1473–1487, 2023.
- [150] A. Li, B. Zheng, G. Pekhimenko, and F. Long, “Automatic horizontal fusion for gpu kernels,” in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 14–27.
- [151] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, “Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 119–130. [Online]. Available: <https://doi-org.offcampus.lib.washington.edu/10.1145/2751205.2751213>
- [152] Y. Sheng, S. Cao, D. Li, B. Zhu, Z. Li, D. Zhuo, J. E. Gonzalez, and I. Stoica, “Fairness in serving large language models,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 965–988. [Online]. Available: <https://www.usenix.org/conference/osdi24/presentation/sheng>
- [153] H. Huang, N. Ardalani, A. Sun, L. Ke, S. Bhosale, H.-H. S. Lee, C.-J. Wu, and B. Lee, “Toward efficient inference for mixture of experts,” in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: <https://openreview.net/forum?id=stXtBqyTWX>
- [154] J. Stojkovic, C. Zhang, Íñigo Goiri, J. Torrellas, and E. Choukse, “Dynamollm: Designing llm inference clusters for performance and energy efficiency,” 2024. [Online]. Available: <https://arxiv.org/abs/2408.00741>
- [155] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, “Fast distributed inference serving for large language models,” 2023. [Online]. Available: <https://arxiv.org/abs/2305.05920>

- [156] 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*, ser. SOSP ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 590–606. [Online]. Available: <https://doi.org/10.1145/3694715.3695964>
- [157] R. Prabhu, A. Nayak, J. Mohan, R. Ramjee, and A. Panwar, “vattention: Dynamic memory management for serving llms without pagedattention,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1133–1150. [Online]. Available: <https://doi.org/10.1145/3669940.3707256>
- [158] C. Hu, H. Huang, L. Xu, X. Chen, J. Xu, S. Chen, H. Feng, C. Wang, S. Wang, Y. Bao, N. Sun, and Y. Shan, “Inference without interference: Disaggregate llm inference for mixed downstream workloads,” 2024. [Online]. Available: <https://arxiv.org/abs/2401.11181>
- [159] A. Jangda, J. Huang, G. Liu, A. H. N. Sabet, S. Maleki, Y. Miao, M. Musuvathi, T. Mytkowicz, and O. Saarikivi, “Breaking the computation and communication abstraction barrier in distributed machine learning workloads,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 402–416. [Online]. Available: <https://doi.org/10.1145/3503222.3507778>
- [160] L.-W. Chang, W. Bao, Q. Hou, C. Jiang, N. Zheng, Y. Zhong, X. Zhang, Z. Song, Z. Jiang, H. Lin, X. Jin, and X. Liu, “Flux: Fast software-based communication overlap on gpus through kernel fusion,” 2024. [Online]. Available: <https://arxiv.org/abs/2406.06858>
- [161] S. Wang, J. Wei, A. Sabne, A. Davis, B. Ilbeyi, B. Hechtman, D. Chen, K. S. Murthy, M. Maggioni, Q. Zhang, S. Kumar, T. Guo, Y. Xu, and Z. Zhou, “Overlap communication with dependent computation via decomposition in large deep learning models,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2022, p. 93–106. [Online]. Available: <https://doi.org/10.1145/3567955.3567959>
- [162] M. Wahib and N. Maruyama, “Scalable kernel fusion for memory-bound gpu applications,” in *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 191–202.

- [163] H. Wu, G. Damos, S. Cadambi, and S. Yalamanchili, “Kernel weaver: Automatically fusing database primitives for efficient gpu computation,” in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 107–118.
- [164] A. Jangda, S. Maleki, M. M. Dehnavi, M. Musuvathi, and O. Saarikivi, “A framework for fine-grained synchronization of dependent gpu kernels,” in *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’24. IEEE Press, 2024, p. 93–105. [Online]. Available: <https://doi.org/10.1109/CGO57630.2024.10444873>
- [165] J. Appleyard and S. Yokim, “Programming Tensor Cores in CUDA 9,” <https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/>, 2017.
- [166] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebron, and S. Sanghai, “GQA: Training generalized multi-query transformer models from multi-head checkpoints,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 4895–4901. [Online]. Available: <https://aclanthology.org/2023.emnlp-main.298/>
- [167] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, “Design and evaluation of the gemtc framework for gpu-enabled many-task computing,” in *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 153–164. [Online]. Available: <https://doi.org/10.1145/2600212.2600228>
- [168] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, “Pagoda: Fine-grained gpu resource virtualization for narrow tasks,” in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 221–234. [Online]. Available: <https://doi.org/10.1145/3018743.3018754>
- [169] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen, “Efficient gpu spatial-temporal multitasking,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 748–760, 2015.
- [170] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, “Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 358–369.

- [171] K. Gupta, J. A. Stuart, and J. D. Owens, “A study of persistent threads style gpu programming for gpgpu workloads,” in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–14.
- [172] “Yi-6B-200K,” <https://huggingface.co/01-ai/Yi-6B-200K>, 2024.
- [173] “Llama-2-7B,” <https://huggingface.co/meta-llama/Llama-2-7b-hf>, 2024.
- [174] “Meta-Llama-3-8B,” <https://huggingface.co/meta-llama/Meta-Llama-3-8B>, 2024.
- [175] “ccdv/arxiv-summarization,” <https://huggingface.co/datasets/ccdv/arxiv-summarization>, 2024.
- [176] “vllm: Easy, fast, and cheap llm serving for everyone,” <https://github.com/vllm-project/vllm>, 2024.
- [177] A. Agrawal, “Merged PR 1865: Critical bug fixes related to sampling,” <https://github.com/microsoft/sarathi-serve/commit/50e59c51b85b1157e001bb8ee7a1b049d551955d#diff-450b0de5cce8a2341140afed859dc5dd3b913fa6e62d27988fccefeacc7b33ec>, 2024.
- [178] LiuXiaoxuanPKU, “Unify the kernel used in flash attention backend,” <https://github.com/vllm-project/vllm/pull/6052>, 2024.
- [179] Y. Ma, P. Gong, T. Wu, J. Yi, C. Yang, C. Li, Q. Peng, G. Xie, Y. Bao, H. Liu, and Y. Xu, “Eliminating data processing bottlenecks in gnn training over large graphs via two-level feature compression,” *Proc. VLDB Endow.*, vol. 17, no. 11, p. 2854–2866, Aug. 2024. [Online]. Available: <https://doi.org/10.14778/3681954.3681968>
- [180] Y. Ma, P. Gong, J. Yi, Z. Yao, C. Li, Y. He, and F. Yan, “Bifeat: Supercharge gnn training via graph feature quantization,” 2023. [Online]. Available: <https://arxiv.org/abs/2207.14696>
- [181] S. Agarwal, C. Yan, Z. Zhang, and S. Venkataraman, “Bagpipe: Accelerating deep recommendation model training,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 348–363. [Online]. Available: <https://doi.org/10.1145/3600006.3613142>
- [182] Y. Lin and V. Grover, “Using cuda warp-level primitives,” <https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/>.

- [183] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, “Open graph benchmark: Datasets for machine learning on graphs,” in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 22 118–22 133. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/fb60d411a5c5b72b2e7d3527cfc84fd0-Paper.pdf
- [184] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, “Collective classification in network data,” *AI Magazine*, vol. 29, no. 3, p. 93, Sep. 2008. [Online]. Available: <https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2157>
- [185] M. Serafini and H. Guan, “Scalable graph neural network training: The case for sampling,” *SIGOPS Oper. Syst. Rev.*, vol. 55, no. 1, 2021. [Online]. Available: <https://doi.org/10.1145/3469379.3469387>
- [186] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7e9bea9-Paper.pdf
- [187] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, “Paragraph: Scaling gnn training on large graphs via computation-aware caching,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 401–415. [Online]. Available: <https://doi.org/10.1145/3419111.3421281>
- [188] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou, “Gnnlab: a factored system for sample-based gnn training over gpus,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 417–434. [Online]. Available: <https://doi-org.offcampus.lib.washington.edu/10.1145/3492321.3519557>
- [189] Y. Wei, M. Langer, F. Yu, M. Lee, J. Liu, J. Shi, and Z. Wang, “A gpu-specialized inference parameter server for large-scale deep recommendation models,” in *Proceedings of the 16th ACM Conference on Recommender Systems*, ser. RecSys ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 408–419. [Online]. Available: <https://doi.org/10.1145/3523227.3546765>
- [190] M. Harris, “How to optimize data transfers in cuda c/c++,” <https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/>.

- [191] “Cuda c++ programming guide: Device memory accesses,” <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory-accesses>.
- [192] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W.-m. Hwu, “Emogi: efficient memory-access for out-of-memory graph-traversal in gpus,” *Proc. VLDB Endow.*, vol. 14, no. 2, p. 114–127, Oct. 2020. [Online]. Available: <https://doi.org/10.14778/3425879.3425883>
- [193] A. Shanbhag, B. W. Yogatama, X. Yu, and S. Madden, “Tile-based lightweight integer compression in gpu,” in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1390–1403. [Online]. Available: <https://doi.org/10.1145/3514221.3526132>
- [194] F. Knorr, P. Thoman, and T. Fahringer, “ndzip-gpu: efficient lossless compression of scientific floating-point data on gpus,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org/10.1145/3458817.3476224>
- [195] A. Weißenberger and B. Schmidt, “Massively parallel inverse block-sorting transforms for bzip2 decompression on gpus,” in *Proceedings of the 53rd International Conference on Parallel Processing*, ser. ICPP ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 856–865. [Online]. Available: <https://doi.org/10.1145/3673038.3673067>
- [196] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtcher, “Mpc: A massively parallel compression algorithm for scientific data,” in *2015 IEEE International Conference on Cluster Computing*, 2015, pp. 381–389.
- [197] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens, “Parallel lossless data compression on the gpu,” in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–9.
- [198] F. Lin, K. Arunruangsirilert, H. Sun, and J. Katto, “Recoil: Parallel rans decoding with decoder-adaptive scalability,” in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 31–40. [Online]. Available: <https://doi.org/10.1145/3605573.3605588>
- [199] “Cuda c++ programming guide: 11.3.2.1. compressible memory,” <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compressible-memory>.

- [200] T. Zhang, M. Hariri, S. Zhong, V. Chaudhary, Y. Sui, X. Hu, and A. Shrivastava, “70% size, 100% accuracy: Lossless LLM compression for efficient GPU inference via dynamic-length float (DFloat11),” in *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026. [Online]. Available: <https://openreview.net/forum?id=xdNAVP7TGy>
- [201] R. Fan, X. Yu, X. Pan, Z. Li, W. Luo, Q. Wang, W. Wang, and X. Chu, “Zipserv: Fast and memory-efficient llm inference with hardware-aware lossless compression,” in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS ’26. New York, NY, USA: Association for Computing Machinery, 2026, p. 2264–2280. [Online]. Available: <https://doi.org/10.1145/3779212.3790250>
- [202] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “Qlora: Efficient finetuning of quantized llms,” in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 10 088–10 115. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/1feb87871436031bdc0f2beaa62a049b-Paper-Conference.pdf
- [203] J. Fang, G. Zhang, J. Han, S. Li, Z. Bian, Y. Li, J. Liu, and Y. You, “A frequency-aware software cache for large recommendation system embeddings,” *arXiv preprint arXiv:2208.05321*, 2022.
- [204] H. Feng, B. Zhang, F. Ye, M. Si, C.-H. Chu, J. Tian, C. Yin, S. Deng, Y. Hao, P. Balaji, T. Geng, and D. Tao, “Accelerating Communication in Deep Learning Recommendation Model Training with Dual-Level Adaptive Lossy Compression ,” in *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2024, pp. 1420–1435. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/SC41406.2024.00095>
- [205] A. Desai and A. Shrivastava, “The trade-offs of model size in large recommendation models : 100gb to 10mb critio-tb dlrn model,” in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 33 961–33 972. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/dbae915128892556134f1c5375855590-Paper-Conference.pdf
- [206] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis, “GRACE: A Compressed Communication Framework for Distributed Machine Learning,” in *ICDCS*, 2021.

- [207] S. Horváth, C.-Y. Ho, v. Horváth, A. N. Sahu, M. Canini, and P. Richtárik, “Natural Compression for Distributed Deep Learning,” in *MSML*, 2022.
- [208] Y. Lin, S. Han, H. Mao, Y. Wang, and W. Dally, “Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training,” in *ICLR*, 2018.
- [209] M. Li, R. B. Basat, S. Vargaftik, C. Lao, K. Xu, M. Mitzenmacher, and M. Yu, “THC: Accelerating Distributed Deep Learning Using Tensor Homomorphic Compression,” in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024. [Online]. Available: <https://www.usenix.org/conference/nsdi24/presentation/li-minghao>
- [210] G. Wang, H. Qin, S. A. Jacobs, X. Wu, C. Holmes, Z. Yao, S. Rajbhandari, O. Ruwase, F. Yan, L. Yang, and Y. He, “ZeRO++: Extremely Efficient Collective Communication for Large Model Training,” in *International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=gx2BT0a9MQ>
- [211] J. Fei, C.-Y. Ho, A. N. Sahu, M. Canini, and A. Sapio, “Efficient Sparse Collective Communication and its application to Accelerate Distributed Deep Learning,” in *SIGCOMM*, 2021.
- [212] C. Yin, J. Jiang, Q. Wang, Z. Mao, and N. Jing, “Spargnn: Efficient joint feature-model sparsity exploitation in graph neural network acceleration,” in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 225–230.
- [213] M. Yoo, J. Song, J. Lee, N. Kim, Y. Kim, and J. Lee, “SGCN: Exploiting Compressed-Sparse Features in Deep Graph Convolutional Network Accelerators,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2023, pp. 1–14. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/HPCA56546.2023.10071102>
- [214] R. Hwang, M. Kang, J. Lee, D. Kam, Y. Lee, and M. Rhu, “Grow: A row-stationary sparse-dense gemm accelerator for memory-efficient graph convolutional neural networks,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 42–55.
- [215] Y. Liu, H. Li, Y. Cheng, S. Ray, Y. Huang, Q. Zhang, K. Du, J. Yao, S. Lu, G. Ananthanarayanan, M. Maire, H. Hoffmann, A. Holtzman, and J. Jiang, “Cachegen: Kv cache compression and streaming for fast large language model serving,” in *Proceedings of the ACM SIGCOMM 2024 Conference*, ser. ACM SIGCOMM ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 38–56. [Online]. Available: <https://doi.org/10.1145/3651890.3672274>

- [216] V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris, “An extended compression format for the optimization of sparse matrix-vector multiplication,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 10, p. 1930–1940, Oct. 2013. [Online]. Available: <https://doi.org/10.1109/TPDS.2012.290>
- [217] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, “Compressed linear algebra for large-scale machine learning,” in *Proceedings of the VLDB Endowment*, vol. 9, no. 12. VLDB Endowment, Aug. 2016, p. 960–971. [Online]. Available: <https://doi.org/10.14778/2994509.2994515>
- [218] F. Li, L. Chen, Y. Zeng, A. Kumar, X. Wu, J. F. Naughton, and J. M. Patel, “Tuple-oriented compression for large-scale mini-batch stochastic gradient descent,” in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1517–1534. [Online]. Available: <https://doi.org/10.1145/3299869.3300070>
- [219] S. Baunsgaard and M. Boehm, “Aware: Workload-aware, redundancy-exploiting linear algebra,” in *Proceedings of the ACM on Management of Data*, vol. 1, no. 1. New York, NY, USA: Association for Computing Machinery, May 2023. [Online]. Available: <https://doi.org/10.1145/3588682>
- [220] U. Köster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloss, R. J. Pai, and N. Rao, “Flexpoint: an adaptive numerical format for efficient training of deep neural networks,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS’17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1740–1750.
- [221] P. Kharya, “Tensorfloat-32 in the a100 gpu accelerates ai training, hpc up to 20x,” <https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/>.
- [222] S. Wang and P. Kanwar, “Bfloat16: The secret to high performance on cloud tpus,” <https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus>.
- [223] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.
- [224] X. Chen, J. Tian, I. Beaver, C. Freeman, Y. Yan, J. Wang, and D. Tao, “Fcbench: Cross-domain benchmarking of lossless compression for floating-point data,” *Proc. VLDB Endow.*, vol. 17, no. 6, p. 1418–1431, may 2024. [Online]. Available: <https://doi.org/10.14778/3648160.3648180>

- [225] Y. Li and J. M. Patel, “Bitweaving: fast scans for main memory data processing,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 289–300. [Online]. Available: <https://doi.org/10.1145/2463676.2465322>
- [226] G. M. Namata, B. London, L. Getoor, and B. Huang, “Query-driven active surveying for collective classification,” in *Workshop on Mining and Learning with Graphs*, 2012. [Online]. Available: <http://lings.cs.umd.edu/basilic/web/Publications/2012/namata:mlg12-wkshp/namata-mlg12.pdf>
- [227] M. Tardy and C. Edwards, “Controlling data movement to boost performance on the nvidia ampere architecture,” <https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/>.
- [228] R. Zou, “Pytorch tutorial: Custom c++ and cuda operators,” https://pytorch.org/tutorials/advanced/cpp_custom_ops.html.
- [229] “Cuda c++ programming guide: Mapped memory,” <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#mapped-memory>.
- [230] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, “Deep graph library: A graph-centric, highly-performant package for graph neural networks,” *arXiv preprint arXiv:1909.01315*, 2019.
- [231] S. Li, H. Liu, Z. Bian, J. Fang, H. Huang, Y. Liu, B. Wang, and Y. You, “Colossal-ai: A unified deep learning system for large-scale parallel training,” in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 766–775. [Online]. Available: <https://doi.org/10.1145/3605573.3605613>
- [232] 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. [Online]. Available: <https://arxiv.org/abs/2205.01068>
- [233] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, P. Tafti, L. Hussenot, P. G. Sessa, A. Chowdhery, A. Roberts, A. Barua, A. Botev, A. Castro-Ros, A. Slone, A. Héliou, A. Tacchetti, A. Bulanova, A. Paterson, B. Tsai, B. Shahriari, C. L. Lan, C. A. Choquette-Choo, C. Crepy, D. Cer, D. Ippolito, D. Reid, E. Buchatskaya, E. Ni, E. Noland, G. Yan, G. Tucker, G.-C. Muraru, G. Rozhdestvenskiy, H. Michalewski, I. Tenney,

- I. Grishchenko, J. Austin, J. Keeling, J. Labanowski, J.-B. Lespiau, J. Stanway, J. Brennan, J. Chen, J. Ferret, J. Chiu, J. Mao-Jones, K. Lee, K. Yu, K. Millican, L. L. Sjoesund, L. Lee, L. Dixon, M. Reid, M. Mikula, M. Wirth, M. Sharman, N. Chinaev, N. Thain, O. Bachem, O. Chang, O. Wahltinez, P. Bailey, P. Michel, P. Yotov, R. Chaabouni, R. Comanescu, R. Jana, R. Anil, R. McIlroy, R. Liu, R. Mullins, S. L. Smith, S. Borgeaud, S. Girgin, S. Douglas, S. Pandya, S. Shakeri, S. De, T. Klimenko, T. Hennigan, V. Feinberg, W. Stokowiec, Y. hui Chen, Z. Ahmed, Z. Gong, T. Warkentin, L. Peran, M. Giang, C. Farabet, O. Vinyals, J. Dean, K. Kavukcuoglu, D. Hassabis, Z. Ghahramani, D. Eck, J. Barral, F. Pereira, E. Collins, A. Joulin, N. Fiedel, E. Senter, A. Andreev, and K. Kenealy, “Gemma: Open models based on gemini research and technology,” 2024. [Online]. Available: <https://arxiv.org/abs/2403.08295>
- [234] A. Bojchevski and S. Günnemann, “Deep gaussian embedding of graphs: Unsupervised inductive learning via ranking,” in *International Conference on Learning Representations*, 2018. [Online]. Available: <https://openreview.net/forum?id=r1ZdKJ-0W>
- [235] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, “Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 982–995.
- [236] “Dlrm checkpoint,” https://catalog.ngc.nvidia.com/orgs/nvidia/teams/dle/models/dlrm_base_tf2_ckpt_ds-criteo-fl15.
- [237] “Terabyte click logs,” <https://labs.criteo.com/2013/12/download-terabyte-click-logs-2/>.
- [238] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.-H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao, “Software-hardware co-design for fast and scalable training of deep learning recommendation models,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 993–1011. [Online]. Available: <https://doi.org/10.1145/3470496.3533727>

- [239] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, “Roformer: Enhanced transformer with rotary position embedding,” *Neurocomputing*, vol. 568, no. C, Feb. 2024. [Online]. Available: <https://doi.org/10.1016/j.neucom.2023.127063>
- [240] F. Knorr, P. Thoman, and T. Fahringer, “Datasets for benchmarking floating-point compressors,” 2020. [Online]. Available: <https://arxiv.org/abs/2011.02849>
- [241] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, “MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 945–960. [Online]. Available: <https://www.usenix.org/conference/nsdi22/presentation/weng>
- [242] A. Kumar, A. Sivasubramaniam, and T. Zhu, “Splitrpc: A Control + Data path splitting rpc stack for ml inference serving,” in *Abstract Proceedings of the 2023 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 13–14. [Online]. Available: <https://doi.org/10.1145/3578338.3593571>
- [243] F. Strati, X. Ma, and A. Klimovic, “Orion: Interference-aware, fine-grained gpu sharing for ml applications,” in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1075–1092. [Online]. Available: <https://doi.org/10.1145/3627703.3629578>
- [244] Y. Gao, Y. He, X. Li, B. Zhao, H. Lin, Y. Liang, J. Zhong, H. Zhang, J. Wang, Y. Zeng, K. Gui, J. Tong, and M. Yang, “An empirical study on low gpu utilization of deep learning jobs,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3597503.3639232>
- [245] P. H. Coppock, B. Zhang, E. H. Solomon, V. Kypriotis, L. Yang, B. Sharma, D. Schatzberg, T. C. Mowry, and D. Skarlatos, “Lithos: An operating system for efficient machine learning on gpus,” in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, ser. SOSP ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1–17. [Online]. Available: <https://doi.org/10.1145/3731569.3764818>
- [246] “NVIDIA Multi-Process Service,” <https://docs.nvidia.com/deploy/mps/index.html>, 2024.

- [247] “NVIDIA Multi-Instance GPU,” <https://www.nvidia.com/en-us/technologies/multi-instance-gpu/>, 2024.
- [248] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, “Gunrock: a high-performance graph processing library on the gpu,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2851141.2851145>
- [249] E. Schmidt and M. Taghavi, “Speeding up data decompression with nvcomp and the nvidia blackwell decompression engine,” Oct 2025. [Online]. Available: <https://developer.nvidia.com/blog/speeding-up-data-decompression-with-nvcomp-and-the-nvidia-blackwell-decompression-engine/>
- [250] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, “Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 1099–1112.
- [251] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, “Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 553–564.
- [252] M. Emani, V. Vishwanath, C. Adams, M. E. Papka, R. Stevens, L. Florescu, S. Jairath, W. Liu, T. Nama, and A. Sujeeth, “Accelerating scientific applications with sambanova reconfigurable dataflow architecture,” *Computing in Science & Engineering*, vol. 23, no. 2, pp. 114–119, 2021.
- [253] M. Chui, E. Hazan, R. Roberts, A. Singla, K. Smaje, A. Sukharevsky, L. Yee, and R. Zemel, “The economic potential of generative ai: The next productivity frontier,” 2023. [Online]. Available: <https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/the-economic-potential-of-generative-AI-the-next-productivity-frontier#introduction>
- [254] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, “Neural message passing for quantum chemistry,” in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.
- [255] Y. Ma, P. Gong, J. Yi, Z. Yao, C. Li, Y. He, and F. Yan, “Bifeat: Supercharge gnn training via graph feature quantization,” 2023. [Online]. Available: <https://arxiv.org/abs/2207.14696>

- [256] A. K. Kamath and A. Basu, “iguard: In-gpu advanced race detection,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 49–65. [Online]. Available: <https://doi.org/10.1145/3477132.3483545>
- [257] S. Pandey, A. K. Kamath, and A. Basu, “Gpm: leveraging persistent memory from a gpu,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 142–156. [Online]. Available: <https://doi.org/10.1145/3503222.3507758>
- [258] —, “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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 688–701. [Online]. Available: <https://doi.org/10.1145/3575693.3575749>
- [259] A. K. Kamath, A. A. George, and A. Basu, “Scord: a scoped race detector for gpus,” in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA ’20. IEEE Press, 2020, p. 1036–1049. [Online]. Available: <https://doi.org/10.1109/ISCA45697.2020.00088>
- [260] I. Cutress, “Amd zen microarchitecture part 2: Extracting instruction-level parallelism,” <https://www.anandtech.com/show/10591/amd-zen-microarchitecture-part-2-extracting-instructionlevel-parallelism/6>, Aug 2016.
- [261] L. Lamport, *LaTeX: A Document Preparation System*, 2nd ed. Reading, Massachusetts: Addison-Wesley, 1994.
- [262] 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)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182. [Online]. Available: <https://www.usenix.org/conference/fast20/presentation/yang>
- [263] 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 *arXiv*. arXiv, 2023. [Online]. Available: <https://arxiv.org/abs/2303.15375>
- [264] P. S. Foundation, “pickle — python object serialization,” <https://docs.python.org/3/library/pickle.html>, October 2023.

- [265] K. Varda, “Protocol buffers: Google’s data interchange format,” *Google Open Source Blog*, Available at least as early as Jul, vol. 72, p. 23, 2008.
- [266] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, “High performance packet processing with flexnic,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 67–81. [Online]. Available: <https://doi.org/10.1145/2872362.2872367>
- [267] V. Petrucci, E. Mirakhur, N. Agarwal, S. W. Lim, V. Tanna, R. Gupta, and M. Wagh, “Cxl memory expansion: A closer look on actual platform,” <https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-memory-expansion-a-close-look-on-actual-platform.pdf>, 2023.
- [268] B. Yan, X. Cheng, B. Jiang, S. Chen, C. Shang, J. Wang, G. Huang, X. Yang, W. Cao, and F. Li, “Revisiting the design of lsm-tree based oltp storage engine with persistent memory,” *Proc. VLDB Endow.*, vol. 14, no. 10, p. 1872–1885, jun 2021. [Online]. Available: <https://doi.org/10.14778/3467861.3467875>
- [269] J. Fang, Y. Yu, C. Zhao, and J. Zhou, “Turbotransformers: an efficient GPU serving system for transformer models,” in *PPoPP ’21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*. ACM, 2021, pp. 389–402. [Online]. Available: <https://doi.org/10.1145/3437801.3441578>
- [270] “Microsoft open sources breakthrough optimizations for transformer inference on GPU and CPU,” <https://cloudblogs.microsoft.com/opensource/2020/01/21/microsoft-onnx-open-source-optimizations-transformer-inference-gpu-cpu/>.
- [271] “Parallel Thread Execution ISA Version 8.5 – Special Registers: %smid,” <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#special-registers-smid>, 2024.
- [272] “Estimating GPT3 API Cost,” <https://medium.com/modern-nlp/estimating-gpt3-api-cost-50282f869ab8>.
- [273] “ONNX Runtime usage at Microsoft,” <https://onnxruntime.ai/about.html>.
- [274] “Model compression and optimization: Why think bigger when you can think smaller?” <https://medium.com/data-science-at-microsoft/model-compression-and-optimization-why-think-bigger-when-you-can-think-smaller-216ec096f68b>.

- [275] “Batch Inference,” <https://mlinproduction.com/batch-inference-vs-online-inference/>.
- [276] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: <https://doi.org/10.18653/v1/n19-1423>
- [277] “American Express Adopts NVIDIA AI to Help Prevent Fraud and Foil Cybercrime,” <https://blogs.nvidia.com/blog/2020/10/05/american-express-nvidia-ai/>.
- [278] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, “Clipper: A {Low-Latency} online prediction serving system,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.
- [279] A. Tirumala, “Iperf: The tcp/udp bandwidth measurement tool,” <http://dast.nlanr.net/Projects/Iperf/>, 1999.
- [280] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.
- [281] A. Williams, N. Nangia, and S. R. Bowman, “A broad-coverage challenge corpus for sentence understanding through inference,” in *Proceedings of NAACL-HLT*, 2018.
- [282] I. Twitter. (2018) Archiveteam json download of twitter stream 2018-04. [Online]. Available: <https://archive.org/details/archiveteam-twitter-stream-2018-04>
- [283] J. R. Gunasekaran, C. S. Mishra, P. Thinakaran, B. Sharma, M. T. Kandemir, and C. R. Das, “Cocktail: A multidimensional optimization for model serving in cloud,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 1041–1057.
- [284] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, “{INFaaS}: Automated model-less inference serving,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 397–411.
- [285] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

- [286] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time,” *Queue*, vol. 14, no. 5, pp. 20–53, 2016.
- [287] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, “Serving {DNNs} like clockwork: Performance predictability from the bottom up,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 443–462.
- [288] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, “Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency,” in *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*, 2017, pp. 109–120.
- [289] “Cocktail repository,” <https://github.com/jashwantraj92/cocktail>.
- [290] “Pytorch version 1.12.0a0+8a1a93a,” <https://github.com/pytorch/pytorch/commit/8a1a93a>.
- [291] “Pytorch AMP,” <https://pytorch.org/docs/stable/amp.html>.
- [292] X. Wang, H. Zhao, and J. Zhu, “Grpc: A communication cooperation mechanism in distributed systems,” *ACM SIGOPS Operating Systems Review*, vol. 27, no. 3, pp. 75–86, 1993.
- [293] “NVIDIA Triton Inference Server,” <https://developer.nvidia.com/nvidia-triton-inference-server>.
- [294] J. Z. Kolter and M. A. Maloof, “Dynamic weighted majority: An ensemble method for drifting concepts,” *The Journal of Machine Learning Research*, vol. 8, pp. 2755–2790, 2007.
- [295] J. Cerquides and R. L. d. Mántaras, “Robust bayesian linear classifier ensembles,” in *European Conference on Machine Learning*. Springer, 2005, pp. 72–83.
- [296] W. H. Beluch, T. Genewein, A. Nürnberger, and J. M. Köhler, “The power of ensembles for active learning in image classification,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 9368–9377.
- [297] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.

- [298] C. Wang, B. Urgaonkar, N. Nasiriani, and G. Kesidis, “Using burstable instances in the public cloud: Why, when and how?” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 1, no. 1, pp. 1–28, 2017.
- [299] “Docker,” <https://www.docker.com/>.
- [300] N.-M. Ho and W.-F. Wong, “Exploiting half precision arithmetic in nvidia gpus,” in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.
- [301] J. Mohan, A. Phanishayee, J. Kulkarni, and V. Chidambaram, “Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 579–596.
- [302] “NVIDIA Triton Dynamic Batching,” https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_configuration.html#dynamic-batcher.
- [303] M. Tan and Q. Le, “Efficientnet: Rethinking model scaling for convolutional neural networks,” in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.
- [304] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.
- [305] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, “Applied machine learning at facebook: A datacenter infrastructure perspective,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.
- [306] “Model ensembles are faster than you think,” <https://ai.googleblog.com/2021/11/model-ensembles-are-faster-than-you.html>.
- [307] “Amazon SageMaker,” <https://aws.amazon.com/sagemaker/>.
- [308] “Azure Machine Learning as a service,” <https://azure.microsoft.com/en-us/products/machine-learning/#product-overview>.
- [309] A. Senior and I. Lopez-Moreno, “Improving dnn speaker independence with i-vector inputs,” in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2014, pp. 225–229.

- [310] M. S. Bartlett, G. Littlewort, M. Frank, C. Lainscsek, I. Fasel, and J. Movellan, “Recognizing facial expression: machine learning and application to spontaneous behavior,” in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*, vol. 2. IEEE, 2005, pp. 568–573.
- [311] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, “Searching for mobilenetv3,” in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1314–1324.
- [312] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, “Recursive deep models for semantic compositionality over a sentiment treebank,” in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013, pp. 1631–1642.
- [313] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, “Huggingface’s transformers: State-of-the-art natural language processing,” *arXiv preprint arXiv:1910.03771*, 2019.
- [314] L. Floridi and M. Chiriatti, “Gpt-3: Its nature, scope, limits, and consequences,” *Minds and Machines*, vol. 30, no. 4, pp. 681–694, 2020.
- [315] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter,” *arXiv preprint arXiv:1910.01108*, 2019.
- [316] A. Polino, R. Pascanu, and D. Alistarh, “Model compression via distillation and quantization,” *arXiv preprint arXiv:1802.05668*, 2018.
- [317] R. T. Mullapudi, S. Chen, K. Zhang, D. Ramanan, and K. Fatahalian, “Online model distillation for efficient video inference,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 3573–3582.
- [318] X. Qiu, L. Zhang, Y. Ren, P. N. Suganthan, and G. Amaratunga, “Ensemble deep learning for regression and time series forecasting,” in *2014 IEEE symposium on computational intelligence in ensemble learning (CIEL)*. IEEE, 2014, pp. 1–6.
- [319] “Microsoft Azure Spot VMs,” <https://azure.microsoft.com/en-in/products/virtual-machines/spot/>.
- [320] “Amazon Web Services Spot VMs,” <https://aws.amazon.com/ec2/spot/>.
- [321] Z. Liu and J. Heer, “The effects of interactive latency on exploratory visual analysis,” *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2122–2131, 2014.

- [322] “Azure VM Pricing,” <https://azure.microsoft.com/en-in/pricing/details/virtual-machines/linux/>.
- [323] C. Wang, M. Li, and A. J. Smola, “Language models with transformers,” *arXiv preprint arXiv:1904.09408*, 2019.
- [324] C. Zhang, M. Yu, W. Wang, and F. Yan, “Enabling cost-effective, slo-aware machine learning inference serving on public cloud,” *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1765–1779, 2020.
- [325] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [326] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.
- [327] N. Schneider, F. Piewak, C. Stiller, and U. Franke, “Regnet: Multimodal sensor registration using deep neural networks,” in *2017 IEEE intelligent vehicles symposium (IV)*. IEEE, 2017, pp. 1803–1810.
- [328] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, “Aggregated residual transformations for deep neural networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500.
- [329] S. Zagoruyko and N. Komodakis, “Wide residual networks,” *arXiv preprint arXiv:1605.07146*, 2016.
- [330] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.
- [331] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, “Electra: Pre-training text encoders as discriminators rather than generators,” *arXiv preprint arXiv:2003.10555*, 2020.
- [332] A. Conneau, K. Khandelwal, N. Goyal, V. Chaudhary, G. Wenzek, F. Guzmán, E. Grave, M. Ott, L. Zettlemoyer, and V. Stoyanov, “Unsupervised cross-lingual representation learning at scale,” in *Annual Meeting of the Association for Computational Linguistics*, 2019.

- [333] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, “Xlnet: Generalized autoregressive pretraining for language understanding,” *Advances in neural information processing systems*, vol. 32, 2019.
- [334] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, “Albert: A lite bert for self-supervised learning of language representations,” *arXiv preprint arXiv:1909.11942*, 2019.
- [335] Z. Dai, G. Lai, Y. Yang, and Q. Le, “Funnel-transformer: Filtering out sequential redundancy for efficient language processing,” *Advances in neural information processing systems*, vol. 33, pp. 4271–4282, 2020.
- [336] P. He, X. Liu, J. Gao, and W. Chen, “Deberta: Decoding-enhanced bert with disentangled attention,” *arXiv preprint arXiv:2006.03654*, 2020.
- [337] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034.
- [338] M. N. Rabe and C. Staats, “Self-attention does not need $o(n^2)$ memory,” 2022.
- [339] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean, “Efficiently scaling transformer inference,” 2022.
- [340] “XFORMERS OPTIMIZED OPERATORS,” <https://facebookresearch.github.io/xformers/components/ops.html>.
- [341] “FlashAttention,” <https://github.com/Dao-AI-Lab/flash-attention>, 2022.
- [342] “FlashInfer: Kernel Library for LLM Serving,” <https://github.com/flashinfer-ai/flashinfer>, 2023.
- [343] “Support KV Partition for BatchPrefill kernel for Paged and Ragged KV-Cache.” <https://github.com/flashinfer-ai/flashinfer/pull/75>, 2024.
- [344] Y. Sun, L. Dong, S. Huang, S. Ma, Y. Xia, J. Xue, J. Wang, and F. Wei, “Retentive network: A successor to transformer for large language models,” 2023.
- [345] M. Artetxe, S. Bhosale, N. Goyal, T. Mihaylov, M. Ott, S. Shleifer, X. V. Lin, J. Du, S. Iyer, R. Pasunuru, G. Anantharaman, X. Li, S. Chen, H. Akin, M. Baines, L. Martin, X. Zhou, P. S. Koura, B. O’Horo, J. Wang, L. Zettlemoyer, M. Diab, Z. Kozareva, and V. Stoyanov, “Efficient large scale language modeling with mixtures of experts,” 2022.

- [346] N. Shazeer, “Fast transformer decoding: One write-head is all you need,” 2019.
- [347] “Tensorrt-llm: A tensorrt toolbox for optimized large language model inference,” <https://github.com/NVIDIA/TensorRT-LLM>, 2023.
- [348] “Lightllm: A light and fast inference service for llm,” <https://github.com/ModelTC/lightllm>, 2023.
- [349] “Faster Transformer,” <https://github.com/NVIDIA/FasterTransformer>, 2024.
- [350] “Performance decay when using paged attention,” <https://github.com/NVIDIA/TensorRT-LLM/issues/75>, 2023.
- [351] “Use optimized kernels for MQA/GQA,” <https://github.com/vllm-project/vllm/issues/1880>, 2023.
- [352] “[BUG] model Yi-34B compat,” <https://github.com/flashinfer-ai/flashinfer/issues/181>.
- [353] “Fix eager mode performance,” <https://github.com/vllm-project/vllm/pull/2377>, 2024.
- [354] “Using NVIDIA’s AI/ML Frameworks for Generative AI on VMware vSphere,” <https://core.vmware.com/blog/using-nvidias-aiml-frameworks-generative-ai-vmware-vmware>.
- [355] “The Secret Sauce behind 100K context window in LLMs: all tricks in one place,” <https://blog.gopenai.com/how-to-speed-up-llms-and-use-100k-context-window-all-tricks-in-one-place-ffd40577b4c>.
- [356] P. Tillet, H. T. Kung, and D. Cox, “Triton: An intermediate language and compiler for tiled neural network computations,” in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 10–19. [Online]. Available: <https://doi.org/10.1145/3315508.3329973>
- [357] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra, “Varuna: scalable, low-cost training of massive deep learning models,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 472–487.

- [358] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, “Pipedream: generalized pipeline parallelism for dnn training,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.
- [359] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-lm: Training multi-billion parameter language models using gpu model parallelism,” *arXiv preprint arXiv:1909.08053*, 2019.
- [360] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, “Gpipe: Efficient training of giant neural networks using pipeline parallelism,” *Advances in neural information processing systems*, vol. 32, 2019.
- [361] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [362] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, “Palm: Scaling language modeling with pathways,” *CoRR*, vol. abs/2204.02311, 2022. [Online]. Available: <https://doi.org/10.48550/arXiv.2204.02311>
- [363] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” *CoRR*, vol. abs/2001.08361, 2020. [Online]. Available: <https://arxiv.org/abs/2001.08361>
- [364] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, “Emergent abilities of large language models,” *Trans. Mach. Learn. Res.*, vol. 2022, 2022. [Online]. Available: <https://openreview.net/forum?id=yzkSU5zdwD>
- [365] OpenAI, “GPT-4 technical report,” *CoRR*, vol. abs/2303.08774, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2303.08774>

- [366] “Character ai,” <https://character.ai>.
- [367] “Bing ai,” <https://www.bing.com/chat>.
- [368] “Nvidia collective communications library (nccl),” <https://developer.nvidia.com/nccl>.
- [369] “Nvidia dgx platform,” <https://www.nvidia.com/en-us/data-center/dgx-platform/>.
- [370] “Komo,” <https://komo.ai/>.
- [371] “You.com,” <https://you.com/>.
- [372] “Perplexity ai,” <https://www.perplexity.ai/>.
- [373] “Google bard,” <https://bard.google.com>.
- [374] “Github copilot,” <https://github.com/features/copilot>.
- [375] “Microsoft copilot,” <https://www.microsoft.com/en-us/microsoft-copilot>.
- [376] “Google duet ai,” <https://workspace.google.com/solutions/ai/>.
- [377] “Replit ghostwriter,” <https://replit.com/site/ghostwriter>.
- [378] “Amazon codewhisperer,” <https://aws.amazon.com/codewhisperer/>.
- [379] “Matrix multiplication background user’s guide,” <https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html>.
- [380] “nanogpt,” <https://github.com/karpathy/nanoGPT>.
- [381] “Faster transformer kernels,” https://github.com/NVIDIA/FasterTransformer/tree/main/src/fastertransformer/kernels/decoder_masked_multihead_attention.
- [382] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, “Llama 2: Open foundation and fine-tuned chat models,” 2023.

- [383] E. Almazrouei, H. Alobeidli, A. Alshamsi, A. Cappelli, R. Cojocaru, M. Debbah, Étienne Goffinet, D. Hesslow, J. Launay, Q. Malartic, D. Mazzotta, B. Noune, B. Pannier, and G. Penedo, “The falcon series of open language models,” 2023.
- [384] A. Cohan, F. Dernoncourt, D. S. Kim, T. Bui, S. Kim, W. Chang, and N. Goharian, “A discourse-aware attention model for abstractive summarization of long documents,” in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 615–621. [Online]. Available: <https://aclanthology.org/N18-2097>
- [385] G. Wang, S. Cheng, X. Zhan, X. Li, S. Song, and Y. Liu, “Openchat: Advancing open-source language models with mixed-quality data,” 2023.
- [386] L. Zheng, W.-L. Chiang, Y. Sheng, T. Li, S. Zhuang, Z. Wu, Y. Zhuang, Z. Li, Z. Lin, E. P. Xing, J. E. Gonzalez, I. Stoica, and H. Zhang, “Lmsys-chat-1m: A large-scale real-world llm conversation dataset,” 2023.
- [387] “Openai gpt-3: Understanding the architecture,” <https://www.theaidream.com/post/openai-gpt-3-understanding-the-architecture>.
- [388] “Text generation inference,” <https://huggingface.co/text-generation-inference>.
- [389] “Deepspeed inference: Multi-gpu inference with customized inference kernels and quantization support,” <https://www.deepspeed.ai/2021/03/15/inference-kernel-optimization.html>.
- [390] J. Li, Y. Jiang, Y. Zhu, C. Wang, and H. Xu, “Accelerating distributed MoE training and inference with lina,” in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 945–959. [Online]. Available: <https://www.usenix.org/conference/atc23/presentation/li-jiamin>
- [391] P. Gao, L. Yu, Y. Wu, and J. Li, “Low latency rnn inference with cellular batching,” in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3190508.3190541>
- [392] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, “Tensorflow-serving: Flexible, high-performance ml serving,” 2017.

- [393] X. Wang, Y. Xiong, Y. Wei, M. Wang, and L. Li, “LightSeq: A high performance inference library for transformers,” in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers (NAACL-HLT)*. Association for Computational Linguistics, Jun. 2021, pp. 113–120.
- [394] I. Beltagy, M. E. Peters, and A. Cohan, “Longformer: The long-document transformer,” 2020.
- [395] G. Bikshandi and J. Shah, “A case study in cuda kernel fusion: Implementing flashattention-2 on nvidia hopper architecture using the cutlass library,” 2023.
- [396] A. Acharya, U. Bondhugula, and A. Cohen, “Effective loop fusion in polyhedral compilation using fusion conflict graphs,” *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 4, p. 1–26, Sep. 2020. [Online]. Available: <http://dx.doi.org/10.1145/3416510>
- [397] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. K. Luk, B. Maher, Y. Pan, C. Puhersch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, and S. Chintala, “Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024.
- [398] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, “Mistral 7b,” *arXiv preprint arXiv:2310.06825*, 2023.
- [399] R. Child, S. Gray, A. Radford, and I. Sutskever, “Generating long sequences with sparse transformers,” 2019.
- [400] “Yi series of large language models trained from scratch by developers at 01.AI.” <https://huggingface.co/01-ai/Yi-34B-200K>.
- [401] “CUDA Toolkit Documentation: Virtual Memory Management,” https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html, 2024.

- [402] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Re, C. Barrett, Z. Wang, and B. Chen, “\$h_2o\$: Heavy-hitter oracle for efficient generative inference of large language models,” in *Conference on Parsimony and Learning (Recent Spotlight Track)*, 2023. [Online]. Available: <https://openreview.net/forum?id=w4IRMAJYPk>
- [403] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, “Arachne: Core-Aware thread management,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 145–160. [Online]. Available: <https://www.usenix.org/conference/osdi18/presentation/qin>
- [404] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, “Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads,” in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’19. USA: USENIX Association, 2019, p. 361–377.
- [405] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, “Coordinated and efficient huge page management with ingens,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 705–721. [Online]. Available: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon>
- [406] A. Panwar, S. Bansal, and K. Gopinath, “Hawkeye: Efficient fine-grained os support for huge pages,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–360. [Online]. Available: <https://doi.org/10.1145/3297858.3304064>
- [407] B. Pichai, L. Hsu, and A. Bhattacharjee, “Architectural support for address translation on gpus: designing memory management units for cpu/gpus with unified address spaces,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 743–758. [Online]. Available: <https://doi.org/10.1145/2541940.2541942>
- [408] Z. Zhang, T. Allen, F. Yao, X. Gao, and R. Ge, “Tunnels for bootlegging: Fully reverse-engineering gpu tlbs for challenging isolation guarantees of nvidia mig,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 960–974. [Online]. Available: <https://doi.org/10.1145/3576915.3616672>

- [409] A. Nayak, P. B., V. Ganapathy, and A. Basu, “(mis)managed: A novel tlb-based covert channel on gpus,” in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 872–885. [Online]. Available: <https://doi.org/10.1145/3433210.3453077>
- [410] S. Shin, M. LeBeane, Y. Solihin, and A. Basu, “Neighborhood-aware address translation for irregular gpu applications,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 352–363.
- [411] D. Hendrycks and K. Gimpel, “Gaussian error linear units (gelus),” 2023.
- [412] A. F. Agarap, “Deep learning using rectified linear units (relu),” 2019.
- [413] R. Abhyankar, Z. He, V. Srivatsa, H. Zhang, and Y. Zhang, “Apiserve: Efficient api support for large-language model inferencing,” *arXiv preprint arXiv:2402.01869*, 2024.
- [414] P. J. Denning, “Virtual memory,” *ACM Comput. Surv.*, vol. 2, no. 3, p. 153–189, sep 1970. [Online]. Available: <https://doi.org/10.1145/356571.356573>
- [415] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, “One-level storage system,” *IRE Transactions on Electronic Computers*, vol. EC-11, no. 2, pp. 223–235, 1962.
- [416] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, “Starcoder: may the source be with you!” 2023.
- [417] C. Guo, R. Zhang, J. Xu, J. Leng, Z. Liu, Z. Huang, M. Guo, H. Wu, S. Zhao, J. Zhao, and K. Zhang, “Gmlake: Efficient and transparent gpu memory defragmentation for large-scale dnn training with virtual memory stitching,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS ’24. New York, NY,

- USA: Association for Computing Machinery, 2024, p. 450–466. [Online]. Available: <https://doi.org/10.1145/3620665.3640423>
- [418] “Add support for small page sizes,” <https://github.com/Dao-AILab/flash-attention/pull/824>, 2024.
 - [419] “Implement Page KV Cache,” <https://github.com/Dao-AILab/flash-attention/commit/54e80a3829c6d2337570d01e78ebd9529c02d342>, 2024.
 - [420] “Refactor Attention Take 2,” <https://github.com/vllm-project/vllm/pull/3462>, 2024.
 - [421] “Use FlashInfer for Decoding,” <https://github.com/vllm-project/vllm/pull/4353>, 2024.
 - [422] “Pascal MMU Format Changes,” <https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf>, 2024.
 - [423] “Separate attention backends,” <https://github.com/vllm-project/vllm/pull/3005/>, 2024.
 - [424] “Yi-34B-200K,” <https://huggingface.co/01-ai/Yi-34B-200K>, 2024.
 - [425] P. B. N. Jawalkar, and A. Basu, “Designing virtual memory system of mcm gpus,” in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’22. IEEE Press, 2023, p. 404–422. [Online]. Available: <https://doi.org/10.1109/MICRO56248.2022.00036>
 - [426] “Custom CUDA kernels for KV cache copy operations,” https://github.com/vllm-project/vllm/blob/main/csrc/cache_kernels.cu, 2024.
 - [427] M. Wu, X. Cheng, O. Padon, and Z. Jia, “A multi-level superoptimizer for tensor programs,” 2024.
 - [428] “Tile primitives for speedy kernels,” <https://github.com/HazyResearch/ThunderKittens>, 2024.
 - [429] “CUDA semantics,” <https://pytorch.org/docs/stable/notes/cuda.html>, 2024.
 - [430] “NVIDIA/cutlass: CUDA Templates for Linear Algebra Subroutines,” <https://github.com/NVIDIA/cutlass>, 2024.
 - [431] “NVIDIA Nsight Compute ,” <https://developer.nvidia.com/nsight-compute>, 2024.

- [432] “Matrix Multiplication Background User’s Guide: 3.2. Wave Quantization,” <https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html#wave-quant>, 2024.
- [433] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, “Flashattention-3: Fast and accurate attention with asynchrony and low-precision,” in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: <https://openreview.net/forum?id=tVConYid20>
- [434] R. Fan, W. Wang, and X. Chu, “Dtc-spm: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 253–267. [Online]. Available: <https://doi.org/10.1145/3620666.3651378>
- [435] “Performance and Tuning,” <https://docs.vllm.ai/en/v0.6.0/models/performance.html>, 2024.
- [436] “Upstream Chunked Prefill,” <https://github.com/vllm-project/vllm/issues/3130>, 2024.
- [437] S. Cao, S. Liu, T. Griggs, P. Schafhalter, X. Liu, Y. Sheng, J. E. Gonzalez, M. Zaharia, and I. Stoica, “Moe-lightning: High-throughput moe inference on memory-constrained gpus,” 2024. [Online]. Available: <https://arxiv.org/abs/2411.11217>
- [438] H. Kang, S. Bharadwaj, J. Hensman, T. Krishna, V. Ruhle, and S. Rajmohan, “Turboattention: Efficient attention approximation for high throughputs llms,” 2024. [Online]. Available: <https://arxiv.org/abs/2412.08585>
- [439] Y. Liu, X. Ao, Z. Qin, J. Chi, J. Feng, H. Yang, and Q. He, “Pick and choose: A gnn-based imbalanced learning approach for fraud detection,” in *Proceedings of the Web Conference 2021*, ser. WWW ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3168–3177. [Online]. Available: <https://doi.org/10.1145/3442381.3449989>
- [440] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, “Graph convolutional neural networks for web-scale recommender systems,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 974–983. [Online]. Available: <https://doi.org/10.1145/3219819.3219890>

- [441] S. Liu, B. Grau, I. Horrocks, and E. Kostylev, “Indigo: Gnn-based inductive knowledge graph completion using pair-wise encoding,” in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 2034–2045. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2021/file/0fd600c953cde8121262e322ef09f70e-Paper.pdf
- [442] P. Bongini, M. Bianchini, and F. Scarselli, “Molecular generative graph neural networks for drug discovery,” *Neurocomputing*, vol. 450, pp. 242–252, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0925231221005737>
- [443] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman, “Mariusggn: Resource-efficient out-of-core training of graph neural networks,” in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 144–161. [Online]. Available: <https://doi.org/10.1145/3552326.3567501>
- [444] “Nvlink and nvlink switch,” <https://www.nvidia.com/en-in/data-center/nvlink/>.
- [445] Z. Qureshi, V. S. Mailthody, I. Gelado, S. W. Min, A. Masood, J. Park, J. Xiong, C. Newburn, D. Vainbrand, I.-H. Chung, M. Garland, W. Dally, and W.-m. Hwu, “Gpu-initiated on-demand high-throughput storage access in the bam system architecture,” in *Proceedings of the Twenty-Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’23, 2023.
- [446] M. Drumond, T. LIN, M. Jaggi, and B. Falsafi, “Training dnns with hybrid block floating point,” in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/6a9aedd6c689c1d0e3b9ccc3ab651bc5-Paper.pdf
- [447] S. Merity, C. Xiong, J. Bradbury, and R. Socher, “Pointer sentinel mixture models,” in *International Conference on Learning Representations*, 2017. [Online]. Available: <https://openreview.net/forum?id=Byj72udxe>
- [448] X. Ning, Z. Lin, Z. Zhou, Z. Wang, H. Yang, and Y. Wang, “Skeleton-of-thought: Prompting LLMs for efficient parallel generation,” in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=mqVgBbNCm9>
- [449] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. R. Narasimhan, “Tree of thoughts: Deliberate problem solving with large language models,” in

- Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: <https://openreview.net/forum?id=5Xc1ecxO1h>
- [450] L. Ye, Z. Tao, Y. Huang, and Y. Li, “ChunkAttention: Efficient self-attention with prefix-aware KV cache and two-phase partition,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 11 608–11 620. [Online]. Available: <https://aclanthology.org/2024.acl-long.623/>
- [451] Z. Ye, R. Lai, B.-R. Lu, C.-Y. Lin, S. Zheng, L. Chen, T. Chen, and L. Ceze, “Cascade inference: Memory bandwidth efficient shared prefix batch decoding,” February 2024. [Online]. Available: <https://flashinfer.ai/2024/02/02/cascade-inference.html>