# Reducing the GPU Memory Bottleneck with Lossless Compression for ML

Aditya K Kamath
University of Washington
Seattle, WA, USA
akkamath@cs.washington.edu

Arvind Krishnamurthy
University of Washington
Seattle, WA, USA
arvind@cs.washington.edu

Marco Canini
KAUST
Thuwal, Saudi Arabia
marco@kaust.edu.sa

Simon Peter
University of Washington
Seattle, WA, USA
simpeter@cs.washington.edu

## Abstract

Machine learning (ML) training and inference often process data sets far exceeding GPU memory capacity, forcing them to rely on PCIe for on-demand tensor transfers, causing critical transfer bottlenecks. Lossy compression has been proposed to relieve bottlenecks but introduces workload-dependent accuracy loss, making it complex or even prohibitive to use in existing ML deployments.

We explore lossless compression as an alternative that avoids this deployment complexity. We identify where lossless compression can be integrated into ML pipelines while minimizing interference with GPU execution. Based on our findings, we introduce Invariant Bit Packing (IBP), a novel lossless compression algorithm designed to minimize data transfer time for ML. IBP identifies and eliminates invariant bits across groups of tensors, improving throughput through GPU-optimized decompression that leverages warp parallelism, low-overhead bit operations, and asynchronous PCIe transfers. We provide easy-to-use APIs, showcasing them by adding IBP support to GNN training, as well as DLRM and LLM inference frameworks. IBP achieves, on average, 74% faster GNN training, 180% faster DLRM embedding lookup, and 25% faster LLM inference.

*CCS Concepts:* • **Information systems → Data compression**; • **Computer systems organization → Neural networks**; **Parallel architectures**.

***Keywords:*** lossless compression, GPU systems, PCIe bottleneck, data movement, machine learning systems, tensor compression, GNN, DLRM, LLM inference

This work is licensed under a Creative Commons Attribution 4.0 International License.
*EUROSYS '26, Edinburgh, Scotland Uk*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2212-7/2026/04
https://doi.org/10.1145/3767295.3803595

**ACM Reference Format:**
Aditya K Kamath, Arvind Krishnamurthy, Marco Canini, and Simon Peter. 2026. Reducing the GPU Memory Bottleneck with Lossless Compression for ML . In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3767295.3803595

## 1 Introduction

Machine learning (ML) models, such as Graph Neural Networks (GNNs), Deep Learning Recommendation Models (DLRMs), and Large Language Models (LLMs) have found widespread use across various domains, such as e-commerce, knowledge graph processing, drug discovery, and fraud detection. Training and inference with these models allow for efficient learning on semi-structured data [18, 53, 55, 89] by processing tensors (i.e., vertex features for GNNs, embedding entries for DLRMs, KV-cache tensors for LLMs) on the scale of hundreds of GBs to TBs, exceeding the memory capacity of available GPUs. To handle this scale, CPU memory [70, 72], disk [76], and networked storage servers [54] are used, which are orders of magnitude larger than GPU memory. Tensors are then transferred on-demand to the GPU.

On-demand data loading causes execution to be bottlenecked by PCIe bandwidth [44, 94]. A significant portion of time is spent waiting for the required data to reach the GPU [92]. Caching [52, 54, 71, 72, 86, 92] and overlapping compute and data transfer provide some benefit, but PCIe bandwidth is over an order of magnitude lower than GPU memory bandwidth [7–9]. Even 10% of cache misses can double the fetch overhead, and if designed inefficiently, PCIe memory fetching can slow down workloads manyfold [58]. Proposals to address this issue typically involve lossy compression. For example, quantization [58, 59, 83] is a popular compaction method reducing 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]. This trade-off is undesirable for commercial applications, due to the risk of revenue loss [12, 14].

We investigate lossless compression as an alternative without the deployment complexity. Making GPU-based lossless compression practical for ML data transfers is a difficult task. CPU-GPU transfer latencies are competitive with GPU decompression latency, making traditional GPU-accelerated compression algorithms heavyweight, and reliance on the host CPU in the critical path can eliminate efficiency gains. In addition, extraneous GPU-resident decompression data structures compete with the model and the input's GPU memory usage, making the use of large buffers for decompression impractical. For example, in GNNs the input tensors of a single batch are on the order of GBs. Reducing GPU memory usage of input requires the batch size to be reduced, necessitating more batches per training epoch [15]. Similarly for LLMs, the GPU memory size limits the context length and batch sizes of requests during inference [43].

We address these challenges by (1) choosing a data layout that enables efficient PCIe transfers directly from the GPU, (2) performing data compression in CPU memory while minimizing GPU-side metadata, and (3) optimizing GPU decompression for just-in-time use. Unlike lossy compression, we avoid any information loss trade-off.

Motivated by these insights, we introduce Invariant Bit Packing (IBP), a GPU-optimized lossless compression algorithm tailored to ML training and inference acceleration. IBP analyzes input data to identify invariant bits shared across tensors. These bits are stored in a fixed-size metadata region (on the order of KBs) within GPU memory and removed from the compressed data items. We provide a parallel implementation of IBP that enables GPU decompression faster than a CPU-to-GPU transfer. IBP utilizes asynchronous, aligned memory accesses [64], low-overhead warp primitives [50], and cheap bitshift operations, achieving low decompression overhead and high throughput (Table 1). IBP makes use of zero-copy [4] to directly fetch and decompress CPU memory from the GPU. The principles we introduce could also apply to disk and network transfers.

Our contributions are as follows:

- We analyze PCIe transfer bottlenecks in GNN, DLRM, and LLM workloads, examine existing lossy and lossless compression, and highlight why prior approaches fail to deliver performance in ML pipelines (§2).
- We introduce Invariant Bit Packing (IBP), an optimized lossless compression algorithm that removes invariant bits across tensors with diverse datatypes (e.g., float32, float16), using minimal metadata (§3–4). Warp-parallel decompression and optimized PCIe data transfer achieve low decompression overhead and high throughput.
- We extend popular ML systems by incorporating IBP into GPU software caches and PCIe traffic paths (§5). Our PyTorch extension and CUDA library simplify adoption across training and inference workloads.
- We evaluate IBP against state-of-the-art GPU-accelerated compression libraries (nvCOMP [6], ndzip-gpu [40]) on real datasets (§6). IBP accelerates GNN training by 74%, DLRM embedding lookup by 180%, and LLM inference by 25% on average on an A100 GPU, while preserving model accuracy. We demonstrate that IBP works on streaming datasets via sampling (§6.1).

We have made our code available at https://github.com/AKKamath/InvariantBitPacking to aid future research.

## 2 Background

Modern ML models increasingly operate on data that significantly exceeds the capacity of available GPU memory. They depend on transferring data from CPU memory over PCIe, introducing critical bottlenecks where PCIe bandwidth (rather than GPU compute) becomes the performance constraint. Latency-hiding techniques like prefetching help hide the transfer latency by fetching data for the next batch while compute for the current batch is ongoing. However, if the transfer latency far exceeds the compute latency, the transfer remains the bottleneck. For example, we evaluate GNN training where the training computation and CPU-GPU transfers are done in parallel, yet training is frequently waiting for the data to arrive (§ 6.2).

We analyze the origin and impact of PCIe transfers in three representative ML workloads—GNN training and DLRM and LLM inference—illustrating why such transfers are a dominant performance constraint (§2.1). We then examine existing techniques to mitigate this overhead, focusing on both lossy and lossless compression, and discuss the trade-offs each approach introduces in ML (§2.2).

### 2.1 PCIe Transfers in Training and Inference

**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 [34] or sparse bitmask vectors [67]. 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 by 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 [68]. 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.

GNNs already make use of a static cache initialized during preprocessing to reduce data movement [52, 71, 72, 86, 92]. The cache has two main components: a data store and a hashmap. All the cached data together make up the data store, possibly discontiguous 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 [52, 58, 71, 92]. Modern GNN frameworks [72, 92] 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, possibly extending to millions of rows, exceeding GPU memory capacity. To handle the capacity requirements, tables can reside in CPU memory, with lookups moving data to the GPU [71]. Each minibatch looks up thousands of table entries—bottlenecking DLRM inference [80]. During inference, embedding tables are static.

**LLM KV-Cache Offloading.** LLMs operate on textual data converted to the form of floating-point tokens (typically 16-bit float, i.e., float16). Each LLM layer uses Attention [75], where each input 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 to determine their relative impact on the current token. Every generated output token repeats this process, being operated on by the K and V of previous tokens.

KV-caching [43] is a technique used by LLMs that avoids recomputing the K and V for each iteration by storing them the first time they are computed, allowing for re-use across output tokens. For modern LLMs reaching context lengths of millions of tokens, the size of the KV-cache far exceeds the size of GPU memory. Offloading [70] is used to increase context lengths by storing the KV-cache in CPU memory, then transferring as needed to the GPU. As the PCIe interconnect is bandwidth-limited, KV-cache offloading has become the critical bottleneck in large-context LLM inference [44, 70]. Sparse KV-caching [44, 93] brings only the critical KV subset onto the GPU, mitigating the bottleneck but not eliminating it, since KV transfers remain on the critical path.

### 2.2 ML Compression Deployment Challenges

Data compression can relieve these transfer bottlenecks, but is challenging to deploy in ML pipelines. Lossy compression 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 paper, we use two terms to describe compression performance:

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

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

**Lossy compression risks model accuracy loss.** In machine learning, quantization [83] 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 [22], which vary across machine learning models [23, 58, 95]. 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 [12, 14].

**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 [35], 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

**Table 1. Average speedup of compressed CPU-to-GPU transfers (best in bold). Existing algorithms generally yield slowdowns. KV-cache shows the range for different LLM layers.**

| Algorithm | Sparse GNN | Dense GNN | DLRM weights | LLM KV-cache (Wikifact-plaintiff) |
|---|---|---|---|---|
| ANS | 2.1 | 0.1 | 0.1 | 0.1 – 0.1 |
| Bitcomp | 3.9 | 0.4 | 0.5 | 0.6 – 0.7 |
| Cascaded | 8.6 | 0.8 | 0.8 | 0.5 – 0.5 |
| GDeflate | 2.1 | 0.3 | 0.5 | 0.1 – 0.8 |
| LZ4 | 2.3 | 0.6 | 0.9 | 0.3 – 2.8 |
| Snappy | 1.6 | 0.4 | 0.7 | 0.2 – 0.9 |
| zStd | 1.8 | 0.1 | 0.3 | 0.1 – 0.9 |
| ndzip | 5.1 | 1.0 | 1.0 | 0.9 – 1.0 |
| **IBP** | **9.7** | **1.2** | **1.1** | **1.0 – 1.1** |

**Table 2. Average space savings (%) across algorithms/data.**

| Algorithm | Sparse GNN | Dense GNN | DLRM weights | LLM KV-cache (Wikifact-plaintiff) |
|---|---|---|---|---|
| ANS | 76.7 | -78.1 | -132.6 | 4.7 – 15.3 |
| Bitcomp | 80.5 | -8.8 | -14.9 | 0 – 0 |
| Cascaded | 89.2 | -1.0 | -2.0 | 0 – 0 |
| GDeflate | 86.9 | -22.7 | -40.8 | 5.7 – 88.6 |
| LZ4 | 89.7 | -0.3 | -1.0 | 0 – 93.3 |
| Snappy | 88.6 | -1.0 | -1.0 | -1.0 – 87.8 |
| zStd | **93.4** | 1.6 | -2.0 | **6.5 – 94.1** |
| ndzip | 62.9 | 6.3 | 4.8 | 0 – 1.0 |
| **IBP** | 92.9 | **10.4** | **8.3** | 2.0 – 9.1 |

**Table 3. Compression time (ms) across algorithms/data.**

| Algorithm | Sparse GNN | Dense GNN | DLRM weights | LLM KV-cache (Wikifact-plaintiff) |
|---|---|---|---|---|
| ANS | 33.1 | 105.4 | 80.4 | 25.2 |
| Bitcomp | 13.0 | 39.0 | 27.9 | 17.4 |
| Cascaded | **12.8** | 33.6 | 26.7 | 35.7 |
| Gdeflate | 68.5 | 482.7 | 316.8 | 394.2 |
| LZ4 | 135.1 | 69.3 | 43.4 | 105.5 |
| Snappy | 133.3 | 81.8 | 55.3 | 165.2 |
| zStd | 53.5 | 292.1 | 266.3 | 121.5 |
| ndzip | 16.4 | **20.7** | **15.0** | **16.5** |
| IBP | 151.5 | 53.2 | 34.2 | 370.2 |

buffer and then reading the relevant code table 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 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 [6], a closed-source NVIDIA library that provides high-performance GPU implementations of popular compression algorithms, as well as ndzip-gpu [40], an open-source GPU-accelerated compression algorithm shown to provide high decompression throughput for floating point data [19] frequently used in ML applications. Table 1 shows the throughput of transferring and decompressing 100,000 compressed tensors, normalized to transferring uncompressed tensors on an A100 GPU, Table 2 shows the space savings, and Table 3 shows the amount of time taken to compress. See § 6 for details on experimental setup and datasets. While future generations of PCIe may improve bandwidth, GPU FLOPS are increasing faster, preserving PCIe as the bottleneck [29].

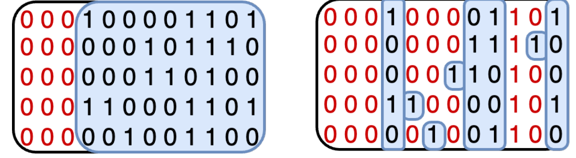


**Figure 1. Conventional bit packing (left) and invariant bit packing (right). Red bits are removed to generate the compressed data shown in blue.**

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. 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 bitpacking achieve a higher transfer speedup. *Cascaded* employs a combination of them and attains the highest speedup among existing algorithms. 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 Bitcomp, Cascaded, and ndzip providing the shortest compression times.

## 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 patterns of invariance found in ML data, which form the basis for our compression (§3.1). Then, we highlight the GPU-specific considerations that influence the design of an effective compression strategy (§3.2).

### 3.1 Invariance in ML Data

From our analysis in §2.2, we find that bit packing performs best among existing algorithms. Bit packing is a family of compression algorithms commonly used in integer compression [48] to remove zero bits as shown in Figure 1 (left). Algorithms such as Cascaded [6] and ndzip [40] 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 number of bits by removing zeroes, 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

**Table 4. Compression of bit packing compared to the fraction of 80p-invariant bits across GNN datasets.**

| Dataset | Sparse | | | Dense | | | DLRM | LLM |
|---|---|---|---|---|---|---|---|---|
| | Pubmed | Citeseer | Cora | Reddit | Product | MAG | | |
| **Bit pack %** | 6.3% | 6.3% | 6.3% | 0% | 0% | 0% | 0% | 0% |
| **Group BP %** | 7.5% | 65.3% | 76% | 0% | 0% | 0% | 0% | 0-6% |
| **Invariant %** | 89.4% | 99.0% | 99.2% | 15.0% | 13.4% | 12.9% | 14.5% | 5-12% |

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.

**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 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.

**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 [31], 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. Similarly sparse datasets have a bias towards zero-valued bits, which we saw in the Pubmed GNN dataset [62]. 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%. When tensors are sampled from a stable distribution, such low-entropy bits remain consistent across tensors. As the number of sampled tensors increases, the empirical probability that such 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 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 tensors.

We empirically demonstrate that ML data frequently has these low-entropy invariant patterns across tensors, (shown in Figure 1 (right)) by evaluating the invariant bits across different ML datasets/weights; Table 4 shows the fraction of bits that are invariant in 80% of tensors or more (80p-invariant). Our notion of invariance is defined by 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 persists across datatype sizes—MAG and LLM use float16, the others use float32. 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 As each compressed tensor varies in size, 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 1, each tensor requires up to four separate bit accesses to decompress.

### 3.2 GPU Architectural Considerations

Unlike CPUs, which can tolerate irregular memory access patterns and rely on complex control logic, GPUs achieve high throughput by executing thousands of lightweight threads in parallel. To fully leverage this parallelism, lossless decompression on GPUs must be designed to minimize memory latency, avoid thread divergence, and maximize coalesced memory accesses. Additionally, GPU memory is a constrained resource, and compression techniques must work within tight memory budgets while maintaining high decompression speed. In this section, we outline the key architectural features and performance considerations of modern GPUs that influence practical lossless compression for ML.

**Hardware and software hierarchies.** The GPU's hardware is arranged in a hierarchy that allows for execution at a scale of hundreds of thousands of parallel threads. The main processor unit of a GPU is a *Streaming Multiprocessor (SM)* [3], with modern GPUs containing around a hundred SMs. Each SM contains an L1 cache and *shared memory* along with its execution units. The shared memory is an L1 cache partition that is user-addressable. The GPU memory is accessed by SMs through the shared L2 cache. Multi-GPU setups are common, where multiple GPUs (typically up to 8) are connected by a high-bandwidth interconnect (e.g., NVLink [10]). This allows the different GPUs to access each other's memory to resemble a scaled-up GPU. GPU programming environments, like CUDA [3], expose a hierarchy of threads that mimic the hardware hierarchy. The smallest unit of execution is a thread. A group of 32 threads makes up a *warp*, which executes in lockstep. A *thread block (TB)* is a

group of warps that share the L1 cache and shared memory. All warps in a TB are guaranteed to be within a single SM.

Leveraging this hierarchy is critical for high-performance GPU programs. *Warp primitives* [50] provide cycle-latency communication and synchronization among threads in the warp. It is critical to dispatch GPU threads in a warp-parallel fashion to leverage these primitives. Each warp can only execute a single common instruction at a time, so GPU programmers must ensure that threads within a warp execute the same code path to maximize throughput.

**Optimizing CPU-to-GPU transfers.** Another critical aspect is the minimization of CPU interactions via the high-latency and bandwidth-constrained PCIe interconnect. When CPU memory must be consulted, there are two methods to transfer data between the CPU and GPU: CPU-initiated and GPU-initiated. In CPU-initiated transfers, the CPU programs the GPU's direct memory access (DMA) engine to transfer data to the GPU [32]. While simpler to program, for small transfers (KBs) the overhead of programming the DMA engine dominates transfer time, lowering performance.

Alternatively, GPU-initiated transfers use GPU warps to directly access and move data from CPU memory to GPU via PCIe. This yields higher performance for small transfers by eliminating DMA and avoiding a PCIe round-trip to the CPU to start the transfer. However, GPU-initiated PCIe transfers need to be carefully aligned to fully leverage the GPU warp parallelism. Aligned memory accesses can be *coalesced* into a single transaction of up to 128B by a hardware coalescer [2]. 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, further reducing throughput. To maximize PCIe throughput, warps in a GPU should generate maximally aligned, 128B transactions when accessing CPU memory [60].

## 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, sparse bitmasks).

We propose *Invariant Bit Packing* (IBP) to achieve these goals. IBP consists of identifying invariant bits in data items, eliminating them, and shifting the remaining bits to reduce the space required for the data item. Decompression involves reading the input bits and reconstructing the original data by reinserting the removed bits. Invariant bit metadata is stored **once** in GPU memory and 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

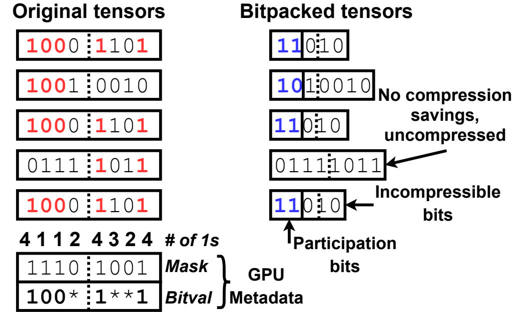



**Figure 2. Invariant bit packing example ($T = 4$, $N = 5$).**

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 threads co-located in the same processor share a common cache (§ 3.2). 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 with minimal overhead 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 [50], which provides cycle latency communication within a warp. By performing this communication, the threads perform a prefix scan to ascertain their starting bit offset, then independently decompress the data items. Once the entire workspace is decompressed, the next segment is read from CPU memory until the entire tensor is decompressed. We make use of asynchronous memory accesses [73] to hide the overhead of decompression by overlapping it with CPU memory accesses. The full process is described in § 4.3.

As the workspace is in the GPU shared memory, 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 within a warp also increases parallelism, while minimizing communication overhead, satisfying our third goal. Finally, IBP operates at a bit level, agnostic to underlying data types, and can be applied across a variety of types, e.g., float32, float16, int, or sparse vectors, fulfilling our last goal.

**Simple ML pipeline integration.** To simplify the integration of IBP in ML pipelines, we provide an easy-to-integrate open-source library. We provide a PyTorch extension [96] 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.

We now look at how we concretely implement IBP, referring to Figure 2 which shows a simple example of IBP for $\frac{T}{N} = \frac{4}{5}$, with invariant bits shown in red. Typical input tensors are KBs in size; for exposition, we use 8-bit tensors.

### 4.1 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 does not interfere with the ML application. For a $\sim$ 180GB dataset, preprocessing and compression only took 84 and 55 seconds respectively.

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 2. For each set bit (i.e., value = 1) 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 as metadata, which are stored in GPU memory due to their small size, avoiding large metadata overheads that scale with input. For ML datasets, this gives good compression ratios (cf. Figure 6.5).

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 and 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. We explore the importance of $T$ further in § 6.5.

### 4.2 Compression

Using *Mask* and *Bitval*, we compress the tensors. Each tensor is first broken into fixed-size chunks (4 bits in the figure), allowing us to parallelize compression across chunks (and later, decompression), 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 the figure). The participation bit indicates whether that chunk is compressed. A chunk can be compressed only 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., as seen in the second chunk of the second tensor.

```
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 3. Pseudocode for IBP tensor compression.**

Similar to preprocessing, compression is performed by the GPU to improve efficiency. The tensors are retained within high-capacity CPU memory and accessed by the GPU using CUDA's zero-copy Unified Virtual Memory feature [4], allowing the GPU direct access to CPU memory. Figure 3 shows the pseudo-code of the two main functions used for compression: pre-calculating 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 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

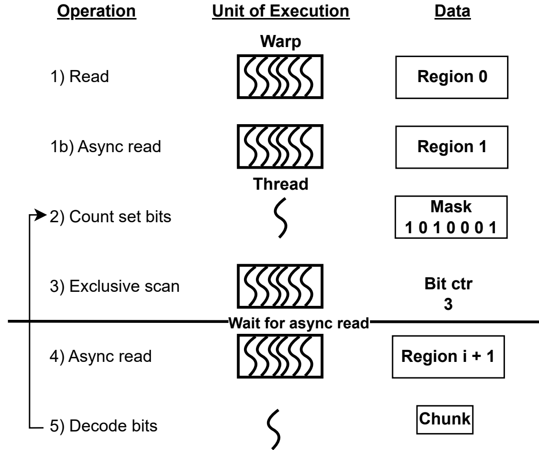



**Figure 4. IBP tensor decompression steps.**

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. In § 5, we shall see how we identify these uncompressed tensors. 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 *bitshift* 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* [50], 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.

## 4.3 Decompression

Decompression is on the critical path of performance and must have minimal overhead. 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 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 (§ 3.2). 1b) Following this, an asynchronous read is initiated proactively. 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 how many 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 reads the required bits 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.

## 4.4 Efficient PCIe Data Transfer

Efficient transfer of data across PCIe is crucial to reducing the overhead of decompression. In Figure 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. Systems like InfiniGen [44] use this method to transfer tensors. *GPU* uses the CUDA zero-copy feature [4] to perform copying from within the GPU. This leverages the GPU's parallelism to transfer the array elements in parallel.

From this investigation, it is clear that initiating copies from the GPU performs better. However, as discussed in § 3.2, 128B size and aligned transfers provide maximum link utilization. Unfortunately, compressed tensors are misaligned

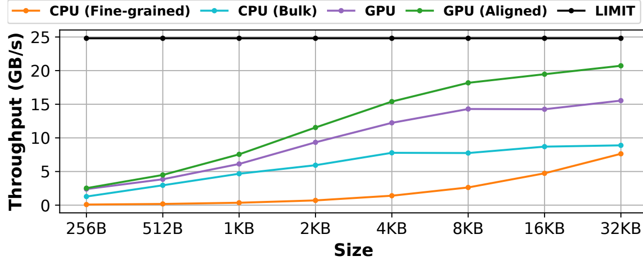



**Figure 5. CPU-to-GPU copy throughput across methods.**

and oddly sized, limiting performance. For this, we use two techniques: ① aligned and ② bounded data transfers.

**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. Similar trends have been observed by other work [60, 64].

**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.

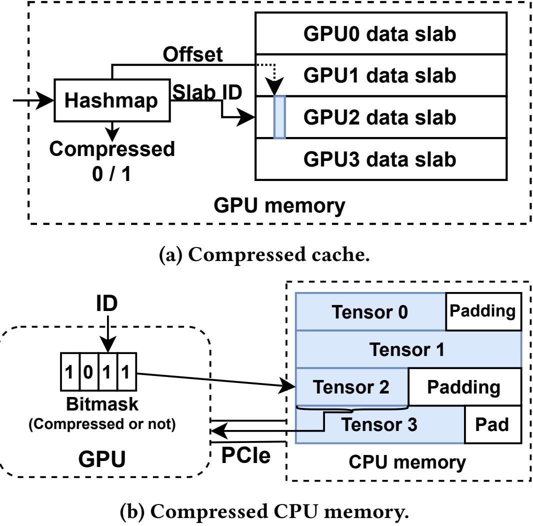
By ensuring the lower and upper bounds are separated by 128B, we copy data into an intermediate buffer, operate on it, and then copy more data while always copying 128B-aligned segments. For example, Step 1 of Figure 4 copies between 1–128 bytes based on the alignment of `src`. Without bounded transfers, if the source was misaligned and we repeatedly copied 128B into a buffer, every copy would be misaligned.

## 5 Using IBP for ML acceleration

We now look at how we integrate IBP compression for ML acceleration, shown in Figure 6. We describe two systems; in the first (Figure 6a), we compress a software cache in GPU memory to fit more tensors in the same space. In the second (Figure 6b), we compress tensors in CPU memory to reduce the amount of data ferried across PCIe.

**Cache compression.** 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 amount of space as the uncompressed version.




**(a) Compressed cache.**

**(b) Compressed CPU memory.**

**Figure 6. Integrating IBP into ML applications.**

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 or otherwise simply transfer 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.

Using these techniques, we can reduce the amount of data transferred across PCIe by 8 - 93% (Table 2).

# 6 Evaluation

In our evaluation, we wish to answer the following questions:

- How well does IBP compress ML datasets versus the state-of-the-art? Does IBP achieve high performance and low overhead for decompression? (§ 6.1)
- What GNN training speedup can IBP provide? (§ 6.2)
- What DLRM inference speedup can IBP provide? (§ 6.3)
- What is the LLM KV offload speedup with IBP? (§ 6.4)
- How do IBP's parameters impact performance? (§ 6.5)

**System configuration.** We evaluate IBP on a system with an A100 GPU with 300 GB of CPU memory. Our system uses 16-lane PCIe Gen 4 as the CPU-GPU interconnect. This provides a theoretical bandwidth of 32GB/s per direction; we experimentally observed ~25GB/s (Figure 5). We use CUDA 11.7, Python 3.8, and Pytorch 1.13.1.

**Baselines.** For compression benchmarking, we use nvCOMP v3.0.1 [6], a closed-source NVIDIA library that provides GPU-accelerated compression algorithms, as well as ndzip-gpu [40], an open-source GPU-accelerated compression algorithm for floating point data. We integrate IBP into two GNN frameworks: DGLv2.1 [78], a popular GNN library, and Legion [72], a state-of-the-art (SOTA) GNN training framework. For DLRM, we integrate IBP into Colossal-AI's CachedEmbeddingBag [26, 47]. For LLM, we integrate IBP into InfiniGen [44], a SOTA KV-cache offload system. We compare IBP against native systems without compression.

## 6.1 IBP (De-)Compression Performance

We start by examining IBP's performance on real datasets, comparing it to existing GPU decompression methods. For compression, we are interested in the compression ratio. For decompression, we care about the throughput for compressed transfers across PCIe as a function of compression.

**Dataset compression ratios.** Table 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 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 datasets, DLRM weights, and LLM KV-cache consist of dense floating-point numbers that are harder to compress. zStd and ndzip can compress some of these. 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 10% for LLM. Overall, IBP's compression across all datasets translates to practical throughput gains.

We also investigate the compression time in Table 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 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. 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, bitshifts) 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%.

**Dataset Sampling.** It may be the case that the entire dataset cannot be preprocessed together, e.g., in distributed deployments or 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, 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 to construct good *Mask* and

**Table 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-Cache | 1.03x | 1.04x | 1.06x | 1.05x | 1.05x | 1.05x |

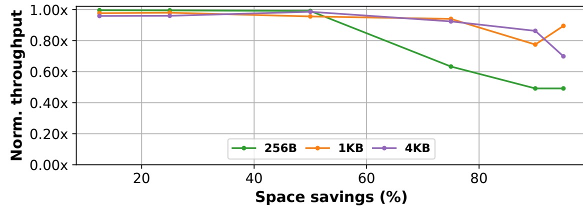



**Figure 7. Decompression throughput versus space savings.**

**Table 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 |

*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.

## 6.2 GNN Training Speedup

**Datasets.** We evaluate 6 public GNN datasets: Pubmed [62], Citeseer [67], Cora [17], Reddit [31], Products [34] and papers of MAG240M [34] (a superset of Papers100M [34]), spanning various tensor and dataset sizes (Table 6).

GNN training is commercially performed on datasets that span 100s of GBs to TBs in size [54]. 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 features 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

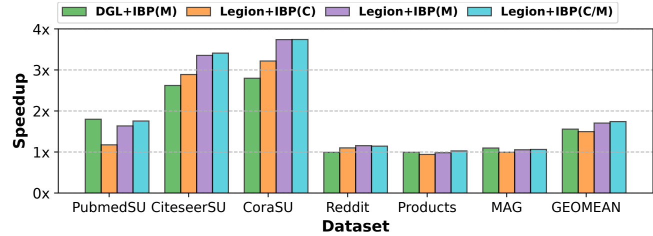



**Figure 8. Average GNN training epoch speedup.**

**Table 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× |

performed by prior work [72, 86]. As the features 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 [54], 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 after transferring them to the GPU. Adding IBP to DGL required fewer than 35 additional lines of Python 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 [31] 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 [72, 86], 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 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 provides good compression, while many of the dataset tensors are not divisible by an 8B chunk size. We verified that IBP's lossless compression did not impact model accuracy.

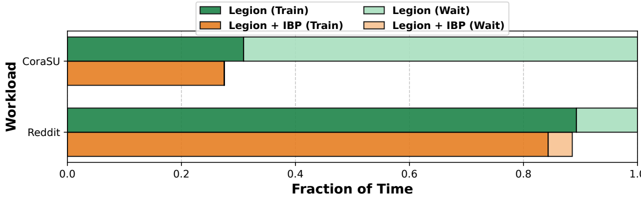



**Figure 9. Breakdown of latency for GNN training epochs.**

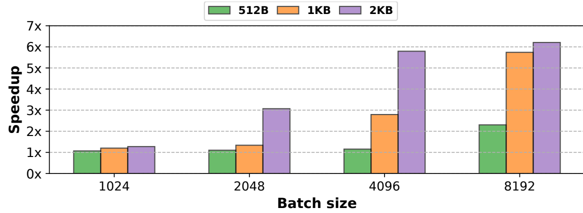



**Figure 10. Normalized DLRM embedding lookup throughput with varying batch and entry sizes.**

**Performance.** Figure 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×, and on average 70%. In general, having both cache compression and PCIe compression improves speedup over each individually.

We also look at how compression influenced the tensor cache used by Legion in Table 7. We see that the increase in capacity of the cache largely correlates with the ratio of compression of the datasets (Table 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 9. Legion uses two processes: one for computationally-heavy training, 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 of speeding up Train time. For CoraSU, IBP decreased training time by 12% versus Legion, due to reduced SM contention for data transfer.

### 6.3 DLRM Inference Transfer Speedup

As discussed in § 2.1, large embedding table lookup and transfer is the bottleneck of DLRM inference [30, 80]. Hence, we evaluate the benefit of IBP on embedding lookups. Besides bookkeeping, adding IBP support required less than 15 lines of Python 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 kept to the default.

**Setup.** We deployed embedding weights from a pre-trained model from NVIDIA [5] trained on the Criteo 1TB dataset [11], 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 [61]. Caching can help maintain frequently accessed data in GPU memory. However, DLRM tables have been shown to reach quite large sizes. As table size increases, cache hit rate decreases. Through our experiment we investigate how well IBP handles DLRM transfers.

**Performance.** Figure 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 (§ 4.4). In the baseline, the CPU performs the table lookups, which consume a significant 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.

### 6.4 LLM KV-Cache Offload Speedup

For large context LLM inference, transferring the entire KV-cache entails significant overhead (§ 2.1). We integrate IBP into InfiniGen [44], a recent LLM KV-cache management framework tailored for long-context inference that improves

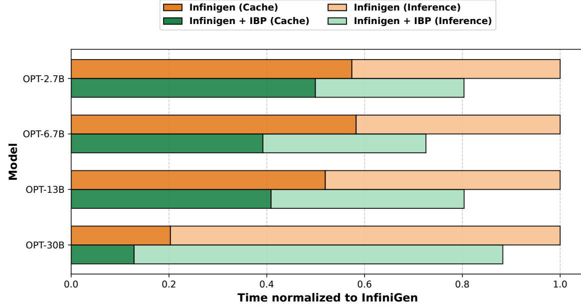



**Figure 11. Normalized LLM inference latency breakdown for 1,920 input and 128 output tokens.**

on FlexGen [70] by 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 [91] with a query of 1,920 input tokens and a batch size of 20, generating 128 output tokens. For LLM inference 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 (cf. § 6.1). 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 11 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 (§ 4.4). 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, which limits the realizable transfer speedup per entry. Newer models like Gemma [74] use larger entries, meaning more gains could be realized.

## 6.5 Performance Sensitivity Studies

We now examine how performance varies with different 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

| 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 12. Space saved with different chunk sizes and invariant threshold fractions $T/N$ for $N = 2.4$M tensors in Products.**

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 12, 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 1 participation bit is needed per byte, leading to 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 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* metadata 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 generic dataset is to use $k$-means clustering. This maximizes the probability 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 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 datasets (Reddit) resemble a normal distribution, where clustering provides very minor benefit, i.e., the peak was 12.7% savings with 477 clusters, 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 [39, 40], 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

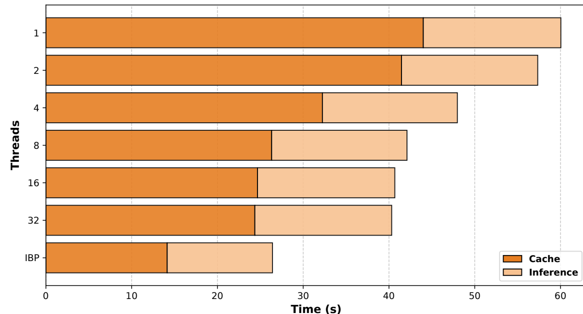


**Figure 13. LLM inference latency with varying CPU threads.**

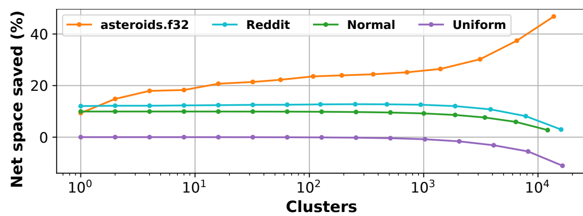


**Figure 14. Clustered compression net space savings.**

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 ∼ 10% net space savings. With 13,725 clusters, the dataset compresses by 58%, yielding a net space saving of 47% (∼ 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 13 we ran the LLM inference experiment varying the number of CPU threads dedicated towards Infinigen.

We see that as the threads increase, the KV cache management performance increases, 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 often infeasible. The CPU performs many duties during ML inference, like fetching inputs [82] or handling RPC calls [42]. Increasing the number of CPUs 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 waiting SMs does not increase contention, while alleviating the PCIe transfer bottleneck.

## 7 Related Work

**GPU-accelerated compression.** Several works propose GPU-based compression [6, 40, 49, 63, 69, 81, 85] focused on exploiting the GPU's parallelism and high throughput to provide high compression and decompression throughput, including bit packing [40, 69]. Other work [13, 66] also examined GPU-accelerated compression in the context of integer-heavy database workloads. NVIDIA A100 and newer GPUs also support generic lossless hardware GPU memory compression [1]. On the opposite end, many projects [20, 57, 65, 87] explore using ML to improve lossy compression performance. Unlike these works, IBP applies bitpacking to ML datasets and CPU-to-GPU transfers, trading off preprocessing overheads for fast, lossless GPU tensor decompression.

**Exploiting sparsity in ML.** Many works try to reduce sparsity in ML applications. Quantization [21, 23, 26, 28, 59, 95] 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 [33, 46, 51, 84] and collective communication [27, 77]. Typically, compression is lossy and can affect the final evaluation results. Various works [36, 88, 90] have also proposed hardware accelerators for GNN training, making use of hardware compression to reduce data accesses. CacheGen [56] investigates the network for LLM KV transfers, proposing dynamic compression to meet latency goals under fluctuating network bandwidths.

Other sets of work [16, 25, 37, 45] look at compressed matrix algebra to improve ML workload performance. FlexPoint [41] proposes a reduced-size floating point data format, similar to TensorFloat32 [38] and BFloat16 [79] 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.

## 8 Conclusion

We propose lossless compression for ML to eliminate invariant bits across tensors, without compromising data fidelity. Our proposal, IBP, uses GPU-accelerated decompression to minimize overhead. Easy integration is demonstrated through IBP-enabled implementations within GNN, DLRM, and LLM frameworks, observing significant performance improvements. This demonstrates IBP's potential to alleviate GPU memory capacity and PCIe bottlenecks, enabling scalable and efficient ML workflows.

## Acknowledgments

We thank our shepherd Yuke Wang and the anonymous reviewers for their valuable feedback on the paper and artifact. We thank Amandio Faustino for his invaluable assistance with system setup, debugging, and facilitating access to the computing environment used in this work. This work is supported by NSF grants 2148209 and 2212193, as well as the University of Washington Center for the Future of Cloud Infrastructure (FOCI). Portions of this work were revised using GenAI tools, in compliance with ACM policy.

## References


[1] Cuda c++ programming guide: Compressible memory. https://docs.nvidia.com/cuda/cuda-c-programming-guide#compressible-memory.

[2] Cuda c++ programming guide: Device memory accesses. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#device-memory-accesses.

[3] Cuda c++ programming guide: Hardware implementation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#hardware-implementation.

[4] Cuda c++ programming guide: Mapped memory. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#mapped-memory.

[5] Dlrm checkpoint. https://catalog.ngc.nvidia.com/orgs/nvidia/teams/dle/models/dlrm_base_tf2_ckpt_ds-criteo-fl15.

[6] nvcomp: High-speed data compression using nvidia gpus. https://developer.nvidia.com/nvcomp.

[7] 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.

[8] Nvidia h100 tensor core gpu. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet.

[9] Nvidia v100 tensor core gpu. https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf.

[10] Nvlink and nvlink switch. https://www.nvidia.com/en-in/data-center/nvlink/.

[11] Terabyte click logs. https://labs.criteo.com/2013/12/download-terabyte-click-logs-2/.

[12] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 802–814, 2021.

[13] Azim Afroozeh, Lotte Felius, and Peter Boncz. Accelerating gpu data processing using fastlanes compression. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*, DaMoN '24, New York, NY, USA, 2024. Association for Computing Machinery.

[14] Saurabh Agarwal, Chengpo Yan, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 348–363, New York, NY, USA, 2023. Association for Computing Machinery.

[15] Saurabh Bajaj, Hojae Son, Juelin Liu, Hui Guan, and Marco Serafini. Graph neural network training systems: A performance comparison of full-graph and mini-batch. In *Proceedings of the VLDB Endowment*, volume 18, page 1196–1209. VLDB Endowment, December 2024.

[16] Sebastian Baunsgaard and Matthias Boehm. Aware: Workload-aware, redundancy-exploiting linear algebra. In *Proceedings of the ACM on Management of Data*, volume 1, New York, NY, USA, May 2023. Association for Computing Machinery.

[17] Aleksandar Bojchevski and Stephan Günnemann. Deep gaussian embedding of graphs: Unsupervised inductive learning via ranking. In *International Conference on Learning Representations*, 2018.

[18] Pietro Bongini, Monica Bianchini, and Franco Scarselli. Molecular generative graph neural networks for drug discovery. *Neurocomputing*, 450:242–252, 2021.

[19] Xinyu Chen, Jiannan Tian, Ian Beaver, Cynthia Freeman, Yan Yan, Jianguo Wang, and Dingwen Tao. Fcbench: Cross-domain benchmarking of lossless compression for floating-point data. In *Proceedings of the VLDB Endowment*, volume 17, page 1418–1431. VLDB Endowment, may 2024.

[20] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Learned image compression with discretized gaussian mixture likelihoods and attention modules. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[21] Aditya Desai and Anshumali Shrivastava. The trade-offs of model size in large recommendation models : 100gb to 10mb criteo-tb dlrm model. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 33961–33972. Curran Associates, Inc., 2022.

[22] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 30318–30332. Curran Associates, Inc., 2022.

[23] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 10088–10115. Curran Associates, Inc., 2023.

[24] Abhinav Dutta, Sanjeev Krishnan, Nipun Kwatra, and Ramachandran Ramjee. Accuracy is not all you need. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 124347–124390. Curran Associates, Inc., 2024.

[25] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. Compressed linear algebra for large-scale machine learning. In *Proceedings of the VLDB Endowment*, volume 9, page 960–971. VLDB Endowment, August 2016.

[26] Jiarui Fang, Geng Zhang, Jiatong Han, Shenggui Li, Zhengda Bian, Yongbin Li, Jin Liu, and Yang You. A frequency-aware software cache for large recommendation system embeddings. *arXiv preprint arXiv:2208.05321*, 2022.

[27] Jiawei Fei, Chen-Yu Ho, Atal N. Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 676–691, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Hao Feng, Boyuan Zhang, Fanjiang Ye, Min Si, Ching-Hsiang Chu, Jiannan Tian, Chunxing Yin, Summer Deng, Yuchen Hao, Pavan Balaji, Tong Geng, and Dingwen 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*, pages 1420–1435, Los Alamitos, CA, USA, November 2024. IEEE Computer Society.

[29] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024.

[30] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean 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)*, pages 982–995, 2020.

[31] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[32] Mark Harris. How to optimize data transfers in cuda c/c++. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.

[33] Samuel Horvóth, Chen-Yu Ho, Ludovit Horvath, Atal Narayan Sahu, Marco Canini, and Peter Richtarik. Natural compression for distributed deep learning. In *Proceedings of Mathematical and Scientific Machine Learning*, volume 190 of *Proceedings of Machine Learning Research*, pages 129–141. PMLR, 15–17 Aug 2022.

[34] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 22118–22133. Curran Associates, Inc., 2020.

[35] David A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the IRE*, volume 40, pages 1098–1101, 1952.

[36] Ranggi Hwang, Minhoo Kang, Jiwon Lee, Dongyun Kam, Youngjoo Lee, and Minsoo 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)*, pages 42–55, 2023.

[37] Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. An extended compression format for the optimization of sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 24(10):1930–1940, October 2013.

[38] Paresh Kharya. Tensorfloat-32 in the a100 gpu accelerates ai training, hpc up to 20x. https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/.

[39] Fabian Knorr, Peter Thoman, and Thomas Fahringer. Datasets for benchmarking floating-point compressors, 2020.

[40] Fabian Knorr, Peter Thoman, and Thomas 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Urs Köster, Tristan J. Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William H. Constable, Oğuz H. Elibol, Scott Gray, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen 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*, NIPS'17, page 1740–1750, Red Hook, NY, USA, 2017. Curran Associates Inc.

[42] Adithya Kumar, Anand Sivasubramaniam, and Timothy Zhu. Splitrpc: A Control + Data path splitting rpc stack for ml inference serving. *SIGMETRICS Perform. Eval. Rev.*, 51(1):13–14, June 2023.

[43] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong 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)*, pages 155–172, Santa Clara, CA, July 2024. USENIX Association.

[45] Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Xi Wu, Jeffrey F. Naughton, and Jignesh M. Patel. Tuple-oriented compression for large-scale mini-batch stochastic gradient descent. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1517–1534, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Minghao Li, Ran Ben Basat, Shay Vargaftik, ChonLam Lao, Kevin Xu, Michael Mitzenmacher, and Minlan Yu. THC: Accelerating Distributed Deep Learning Using Tensor Homomorphic Compression. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[47] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 766–775, New York, NY, USA, 2023. Association for Computing Machinery.

[48] Yinan Li and Jignesh M. Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 289–300, New York, NY, USA, 2013. Association for Computing Machinery.

[49] Fangzheng Lin, Kasidis Arunruangsirilert, Heming Sun, and Jiro Katto. Recoil: Parallel rans decoding with decoder-adaptive scalability. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 31–40, New York, NY, USA, 2023. Association for Computing Machinery.

[50] Yuan Lin and Vinod Grover. Using cuda warp-level primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[51] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *International Conference on Learning Representations*, 2018.

[52] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 401–415, New York, NY, USA, 2020. Association for Computing Machinery.

[53] Shuwen Liu, Bernardo Grau, Ian Horrocks, and Egor Kostylev. Indigo: Gnn-based inductive knowledge graph completion using pair-wise encoding. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 2034–2045. Curran Associates, Inc., 2021.

[54] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong 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)*, pages 103–118, Boston, MA, April 2023. USENIX Association.

[55] Yang Liu, Xiang Ao, Zidi Qin, Jianfeng Chi, Jinghua Feng, Hao Yang, and Qing He. Pick and choose: A gnn-based imbalanced learning approach for fraud detection. In *Proceedings of the Web Conference 2021*, WWW '21, page 3168–3177, New York, NY, USA, 2021. Association for Computing Machinery.

[56] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 38–56, New York, NY, USA, 2024. Association for Computing Machinery.

[57] Guo Lu, Wanli Ouyang, Dong Xu, Xiaoyun Zhang, Chunlei Cai, and Zhiyong Gao. Dvc: An end-to-end deep video compression framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[58] Yuxin Ma, Ping Gong, Tianming Wu, Jiawei Yi, Chengru Yang, Cheng Li, Qirong Peng, Guiming Xie, Yongcheng Bao, Haifeng Liu, and Yinlong Xu. Eliminating data processing bottlenecks in gnn training over large graphs via two-level feature compression. In *Proceedings of the VLDB Endowment*, volume 17, page 2854–2866. VLDB Endowment, August 2024.

[59] Yuxin Ma, Ping Gong, Jun Yi, Zhewei Yao, Cheng Li, Yuxiong He, and Feng Yan. Bifeat: Supercharge gnn training via graph feature quantization. *arXiv preprint arXiv:2207.14696*, 2023.

[60] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. Emogi: efficient memory-access for out-of-memory graph-traversal in gpus. In *Proceedings of the VLDB Endowment*, volume 14, page 114–127. VLDB Endowment,

October 2020.

[61] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware codesign for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 993–1011, New York, NY, USA, 2022. Association for Computing Machinery.

[62] Galileo Mark Namata, Ben London, Lise Getoor, and Bert Huang. Query-driven active surveying for collective classification. In *Workshop on Mining and Learning with Graphs*, 2012.

[63] Ritesh A. Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D. Owens. Parallel lossless data compression on the gpu. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, 2012.

[64] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seung Won Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wenmei Hwu. Gpu-initiated on-demand high-throughput storage access in the BaM system architecture. In *Proceedings of the Twenty-Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '23, 2023.

[65] Oren Rippel and Lubomir Bourdev. Real-time adaptive image compression. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2922–2930. PMLR, 06–11 Aug 2017.

[66] Eyal Rozenberg and Peter Boncz. Faster across the pcie bus: a gpu library for lightweight decompression: including support for patched compression schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, DAMON '17, New York, NY, USA, 2017. Association for Computing Machinery.

[67] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93, Sep. 2008.

[68] Marco Serafini and Hui Guan. Scalable graph neural network training: The case for sampling. *SIGOPS Oper. Syst. Rev.*, 55(1), 2021.

[69] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. Tile-based lightweight integer compression in gpu. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1390–1403, New York, NY, USA, 2022. Association for Computing Machinery.

[70] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31094–31116. PMLR, 23–29 Jul 2023.

[71] Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. Ugache: A unified gpu cache for embedding-based deep learning. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 627–641, New York, NY, USA, 2023. Association for Computing Machinery.

[72] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, and Fei Wu. Legion: Automatically pushing the envelope of Multi-GPU system for Billion-Scale GNN training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 165–179, Boston, MA, July 2023. USENIX Association.

[73] Matthieu Tardy and Carter 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/.

[74] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikuła, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran, Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. Gemma: Open models based on gemini research and technology, 2024.

[75] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[76] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 144–161, New York, NY, USA, 2023. Association for Computing Machinery.

[77] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Xiaoxia Wu, Connor Holmes, Zhewei Yao, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. ZeRO++: Extremely Efficient Collective Communication for Large Model Training. In *International Conference on Learning Representations*, 2024.

[78] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[79] Shibo Wang and Pankaj 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.

[80] Yingcan Wei, Matthias Langer, Fan Yu, Minseok Lee, Jie Liu, Ji Shi, and Zehuan Wang. A gpu-specialized inference parameter server for

large-scale deep recommendation models. In *Proceedings of the 16th ACM Conference on Recommender Systems*, RecSys '22, page 408–419, New York, NY, USA, 2022. Association for Computing Machinery.

[81] André Weißenberger and Bertil Schmidt. Massively parallel inverse block-sorting transforms for bzip2 decompression on gpus. In *Proceedings of the 53rd International Conference on Parallel Processing*, ICPP '24, page 856–865, New York, NY, USA, 2024. Association for Computing Machinery.

[82] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu 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)*, pages 945–960, Renton, WA, April 2022. USENIX Association.

[83] B. Widrow, I. Kollar, and Ming-Chang Liu. Statistical theory of quantization. *IEEE Transactions on Instrumentation and Measurement*, 45(2):353–361, 1996.

[84] Hang Xu, Chen-Yu Ho, Ahmed M. Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. GRACE: A Compressed Communication Framework for Distributed Machine Learning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 561–572, 2021.

[85] Annie Yang, Hari Mukka, Farbod Hesaaraki, and Martin Burtscher. Mpc: A massively parallel compression algorithm for scientific data. In *2015 IEEE International Conference on Cluster Computing*, pages 381–389, 2015.

[86] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: a factored system for sample-based gnn training over gpus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 417–434, New York, NY, USA, 2022. Association for Computing Machinery.

[87] Ruihan Yang and Stephan Mandt. Lossy image compression with conditional diffusion models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 64971–64995. Curran Associates, Inc., 2023.

[88] Chen Yin, Jianfei Jiang, Qin Wang, Zhigang Mao, and Naifeng 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)*, pages 225–230, 2024.

[89] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, page 974–983, New York, NY, USA, 2018. Association for Computing Machinery.

[90] Mingi Yoo, Jaeyong Song, Jounghoo Lee, Namhyung Kim, Youngsok Kim, and Jinho Lee. SGCN: Exploiting Compressed-Sparse Features in Deep Graph Convolutional Network Accelerators . In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–14, Los Alamitos, CA, USA, March 2023. IEEE Computer Society.

[91] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[92] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. Ducati: A dual-cache training system for graph neural networks on giant graphs with the gpu. In *Proceedings of the ACM on Management of Data*, volume 1, New York, NY, USA, June 2023. Association for Computing Machinery.

[93] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2o: heavy-hitter oracle for efficient generative inference of large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[94] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Mingxia Li, Fan Yang, Qianxi Zhang, Binyang Li, Yuqing Yang, Lili Qiu, Lintao Zhang, and Lidong Zhou. Silod: A co-design of caching and scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 883–898, New York, NY, USA, 2023. Association for Computing Machinery.

[95] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 196–209, 2024.

[96] Richard Zou. Pytorch tutorial: Custom c++ and cuda operators. https://pytorch.org/tutorials/advanced/cpp_custom_ops.html.

# A Artifact Appendix

## A.1 Abstract

We provide the source code and setup necessary for Invariant Bit Packing library alongside applications and microbenchmarks necessary to replicate the results from this paper.

The library contains the CUDA backend and Python module for IBP. The applications are GNN (DGL [78] and Legion [72]), DLRM [5], and LLM [44]. All necessary datasets are directly downloaded by our provided scripts.

## A.2 Description & Requirements

Our requirements are primarily hardware, specifically an NVIDIA A100 GPU. The software requirements are handled through Docker.

### A.2.1 How to access.

***GitHub:*** https://github.com/AKKamath/InvariantBitPacking

***Zenodo:*** https://zenodo.org/records/18869047

**A.2.2 Hardware dependencies.** An NVIDIA A100 GPU and minimum 300 GB CPU memory, 1 TB disk space.

**A.2.3 Software dependencies.** The software dependencies will get resolved in our Docker installation.

We tested on Ubuntu 22.04 OS with CUDA 11.7, Python 3.8, and Pytorch 1.13.1.

**A.2.4 Benchmarks.** All required benchmarks are downloaded directly by the scripts in the artifact. These are: Pubmed [62], Citeseer [67], Cora [17], Reddit [31], OGB-Products [34], and MAG [34] GNN datasets; DLRM precomputed weights [5]; and OPT-2.7B, 6.7B, 13B, and 30B LLM models [91].

## A.3 Setup

The repository contains a README.md file with instructions on setup. That is the definitive instruction on setting

up, and may be more up-to-date than instructions given here.

Download repository and datasets:

```
git clone https://github.com/AKKamath/
    InvariantBitPacking.git
cd InvariantBitPacking
git submodule update --init --recursive

make download_dlrm # 16GB download
make download_llm # 8GB download
```

We provide two methods of installing and testing, Docker and manual installation. We highly recommend using the Docker approach, as the manual installation can run into issues with library version mismatches causing errors.

**A.3.1 Docker.** We provide a docker image with all dependencies pre-installed. You need to have Docker and NVIDIA Container Toolkit installed on your system. You can build the docker image and launch as follows:

```
# Build the local docker image
docker build -t ibp-image -f Dockerfile .
# Run the docker image
docker run --gpus all -it -p 8181:8181 \
   --rm --ipc=host --cap-add=SYS_ADMIN ibp-image
```

**A.3.2 Manual installation.** For manual installation, we download the IBP repository to the home directory and install it. We use Anaconda for fetching the appropriate versions of CUDA, Python, and PyTorch. This can take up to 2 hours.

```
# If conda not installed:
make install_miniconda
# Setup:
make create_env
conda activate ibp
# If system does not have CUDA 11.7 installed, the
    below installs it in conda.
make install_cuda
conda env config vars set CUDA_HOME="${CONDA_PREFIX}"
conda env config vars set CUDA_TOOLKIT_ROOT_DIR="${
    CONDA_PREFIX}"
conda env config vars set CUDACXX="${CONDA_PREFIX}/
    bin/nvcc"
conda env config vars set PATH=$CONDA_PREFIX/bin:
    $PATH
conda env config vars set LD_LIBRARY_PATH=
    $CONDA_PREFIX/lib:$LD_LIBRARY_PATH
# If CUDA 11.7 is already installed, continue from
    here.
make install_deps
conda deactivate
conda activate ibp
# Install NVComp, NDZip, Legion, DGL, ColossalAI, IBP
make install
# Download GNN dataset (dependent on DGL)
make download_gnn
```

### A.4 Evaluation workflow

**A.4.1 Major Claims.** The claims made in the paper follows:

- (C1): IBP achieves the best average speedup of compressed CPU-to-GPU transfers across all datasets as shown in Table 1.
- (C2): IBP achieves speedups for DGL and Legion for GNN training as shown in Figure 8.
- (C3): IBP improves throughput for DLRM embedding lookups as shown in Figure 10.
- (C4): IBP reduces inference latency for InfiniGen as shown in Figure 11.

**A.4.2 Experiments.** The instructions below explain how to run the experiments to evaluate the major claims listed prior. Due to system differences and runtime variation, the exact performance numbers may not match those seen in the paper, but the relative trends should hold true.

**Experiment (E1):** *Comparison of different compression algorithm performances.*
This experiment compares the relative speed and compression ratios of the different algorithms on the different datasets. To run this experiment type '`make nvcomp_comparison`'. The results will be found in results/nvcomp_comparison.log; you can run '`cat results/nvcomp_comparison.log`' to output to terminal. **Note that the output provides the details for every dataset individually.** For the sake of space, Table 1 and Table 2 considers the average of Pubmed, Citeseer, and Cora as 'Sparse GNN', and the average of Reddit, Product and MAG as 'Dense GNN'. This should verify claim C1 by generating results similar to Table 1 and Table 2.

**Experiment (E2):** *GNN training performance.*
This experiment compares the runtime of DGL and Legion with and without IBP enabled. To run this experiment type '`make gnn`'. The results will be found in results/gnn _perf.log. You can run '`cat results/gnn_perf.log`' to output to terminal, then copy-paste into your favorite graph plotting software (e.g., Excel, Google Sheets) to plot into a bar chart. This should verify claim C2 with results similar to Figure 8.

**Experiment (E3):** *DLRM lookup throughput.*
This experiment compares the lookup throughput of DLRM embeddings with varying batch and entry sizes. To run this experiment type '`make dlrm`'. The results will be found in results/dlrm_perf.log. You can run '`cat results/dlrm_perf.log`' to output to terminal, then copy-paste into your favorite graph plotting software (e.g., Excel, Google Sheets) to plot into a bar chart. This should verify claim C3 with results similar to Figure 10.

**Experiment (E4):** *LLM inference latency.*
This experiment compares the inference latency of InfiniGen with and without IBP enabled. To run this experiment type '`make llm`'. The results will be found in results/llm_latency.log.

You can run '`cat results/llm_latency.log`' to output to terminal, then copy-paste into your favorite graph plotting software (e.g., Excel, Google Sheets) to plot into a bar chart. This should verify claim C4 with results similar to Figure 11.

### A.5 Notes on Reusability

The paper proposes a methodology on transferring compressed data during execution time of ML models to reduce the transfer overhead. We provide a library that demonstrates this concretely which can be used out-of-the-box. The include/ folder in our repository contains the header-only CUDA portion of our code, which can be integrated into frameworks relying on CUDA. We also have a Pytorch module which can be installed with:

```
pip install -e .
```

Once installed, the module can be imported by putting 'import ibp' in your Python file.