# LIA: A Single-GPU LLM Inference Acceleration with Cooperative AMX-Enabled CPU-GPU Computation and CXL Offloading


Hyungyo Kim
University of Illinois at Urbana-Champaign
Champaign, Illinois, USA
hyungyo2@illinois.edu

Nachuan Wang
University of Illinois at Urbana-Champaign
Champaign, Illinois, USA
nachuan3@illinois.edu

Qirong Xia
University of Illinois at Urbana-Champaign
Champaign, USA
qirongx2@illinois.edu

Jinghan Huang
University of Illinois at Urbana-Champaign
Champaign, USA
jinghan4@illinois.edu

Amir Yazdanbakhsh
Google DeepMind
Mountain View, USA
ayazdan@google.com

Nam Sung Kim
University of Illinois at Urbana-Champaign
Champaign, USA
nam.sung.kim@gmail.com



## Abstract

The limited memory capacity of single GPUs constrains large language model (LLM) inference, necessitating cost-prohibitive multi-GPU deployments or frequent performance-limiting CPU-GPU transfers over slow PCIe. In this work, we first benchmark recent Intel CPUs with Advanced Matrix Extensions (AMX), including 4th generation (Sapphire Rapids) and 6th generation (Granite Rapids) Xeon Scalable Processors, demonstrating matrix multiplication throughput of 20 TFLOPS and 40 TFLOPS, respectively—comparable to some recent GPUs. These findings unlock more extensive computation offloading to CPUs, reducing CPU-GPU transfers and alleviating throughput bottlenecks compared to prior-generation CPUs. Building on these insights, we design LIA, a single-GPU LLM inference acceleration framework leveraging cooperative AMX-enabled CPU-GPU computation and CXL offloading. LIA systematically offloads computation to CPUs, optimizing both latency and throughput. The framework also introduces a memory-offloading policy that seamlessly integrates affordable CXL memory with DDR memory to enhance performance in throughput-driven tasks. On Saphhire Rapids (Granite Rapids) systems with a single H100 GPU, LIA achieves up to 5.1× (19×) lower latency and 3.7× (5.1×) higher throughput compared to the latest single-GPU offloading framework. Furthermore, LIA deploying CXL offloading yields an additional 1.5× throughput improvement over LIA using only DDR memory with a 1.8× increase in maximum batch size (900→1.6K).



## CCS Concepts

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Hardware** → **Emerging architectures**; • **General and reference** → **Design**.



Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).


*ISCA '25, Tokyo, Japan*


© 2025 Copyright held by the owner/author(s).


ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731092

## Keywords

LLM Inference, GPU-CPU Cooperative Computing, AMX, CXL

**ACM Reference Format:**
Hyungyo Kim, Nachuan Wang, Qirong Xia, Jinghan Huang, Amir Yazdanbakhsh, and Nam Sung Kim. 2025. LIA: A Single-GPU LLM Inference Acceleration with Cooperative AMX-Enabled CPU-GPU Computation and CXL Offloading. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731092


## 1 Introduction

Large Language Models (LLMs) have unleashed transformative potential across a vast array of applications, spanning natural language processing [32], conversational AI [52], image and video synthesis [27], and document processing and summarization [39]. However, this unprecedented capability comes with a significant cost: *the relentless expansion of model parameters.* Recent models such as GPT-4 [36], Gemini [20], Llama-3 [16], PaLM [14], Bloom [28], and OPT [53] are now engineered with hundreds of billions of parameters, and this upward trajectory appears far from plateauing [26]. This increase in the number of parameters introduces a major technical challenge: *storing both the model parameters and intermediate values—such as the Key-Value (KV) cache and activations—has become infeasible on a single GPU.* Even the latest high-end GPUs (*i.e.,* H100), equipped with up to 94 GB of HBM memory, struggle to meet these storage demands during inference. Given the limited capacity of single GPUs, recent work has turned to multi-GPU deployments along with model parallelism [30, 35, 44]. Yet, this approach is financially prohibitive and operationally complex; For example, deploying the 175 billion-parameter OPT model [53] for inference requires at least five NVIDIA H100 GPUs, with a total cost of $150,000. Therefore, scaling by increasing GPU count is simply not a viable option in cost-efficient inference scenarios.

Various compression techniques, such as quantization [15, 19, 50], pruning [18, 31, 47], and model distillation [11, 21, 25], have been proposed to alleviate the cost burden of LLM scalability. Although these methods reduce memory requirements, they often come at the expense of model accuracy and still require multiple

GPUs.[1] An alternative direction is system-level offloading, where model parameters are stored in large CPU memory and transferred to GPUs on demand [13, 40, 43]. Nonetheless, this approach faces its own bottlenecks because of the limited PCIe bandwidth (*i.e.,* H100 based on PCIe 5.0 has 64 GB/s), which slows CPU-GPU transfers, leading to considerable inference latency increase.[2]

To mitigate the significant overhead of large data transfers, several approaches have proposed selectively offloading certain layers or computation to the CPU [43, 45]. However, the effectiveness of these CPU-GPU collaborative frameworks has been constrained by the CPU's limited compute throughput, which is 100× lower than that of a high-end GPU (§4). For example, FlexGen [43] and FastDecode [23] offload only the least compute-intensive sublayer to the CPU, while PowerInfer [45] requires model adaptation to a limited set of models (with ReLU activation) to minimize the number of cold neurons processed by the CPU which often degrades the accuracy of the model significantly. Adding to the complexity, LLM inference operates across varying batch sizes depending on the application requirements. Small-batch, low-latency inference is indispensable for user-facing applications like virtual assistants [52] and search engines [46], where maintaining a fast response time directly impacts user experience. In contrast, large-batch, high-throughput processing is essential for latency-insensitive applications, such as benchmarking [29], information extraction [34], and data wrangling [33], where processing a large corpus of data takes precedence over immediate response. Despite these diverse requirements, prior approaches neglect the varying compute and data transfer requirements with different batch sizes and also sequence lengths, leading to the suboptimality of a single fixed compute-offloading policy.

To address these challenges, we present **LIA** (**L**LM **I**nference **A**cceleration), a framework that accelerates a single-GPU LLM inference across both small- and large-batch scenarios using AMX and CXL technologies. LIA brings forth three main contributions:

**Comprehensive analysis of AMX matrix-multiplication performance (§4).** Since the introduction of Intel's 4th generation Xeon Scalable Processors, codenamed Sapphire Rapids (SPR) [8], in 2023, Intel Xeon CPUs have integrated AMX, a built-in matrix-multiplication accelerator. The performance of AMX scales with the number of cores, and the latest 6th generation Xeon Scalable Processors, codenamed Granite Rapids (GNR) [9], more than doubling the maximum core count than the previous generations, significantly elevated AMX's performance potential. Recognizing AMX's potential for accelerating LLMs, we conduct a series of matrix-multiplication microbenchmarks on SPR and GNR with realistic matrix and vector sizes derived from LLM workloads. Our evaluation reveals that AMX-enabled SPR (GNR) can deliver up to 4.5× (9×) higher GEMM throughput than AVX512, which other frameworks [43, 45] have used for CPU-offloaded computation, and achieves up to 11% (22%) and 5% (10%) of the GEMM throughput of A100 and H100, respectively.[3] Additionally, AMX-enabled SPR (GNR) achieves 38% (44%) and 35% (41%) of GEMV throughput compared to A100 and H100, respectively. These results underscore the potential of AMX-equipped CPUs for efficient compute offloading, expanding the role of the CPU beyond the scope of recent frameworks [43, 45].

**AMX-driven CPU-GPU synergy for LLM inference acceleration (§5).** Building on the insights from our benchmarking, we develop LIA to systematically orchestrate CPU-GPU synergy by leveraging the relatively high throughput of AMX. This enables productive offloading of a larger portion of LLM computation to the CPU, optimizing LLM inference performance. LIA consists of two main components: **(C1)** the algorithm front-end, which systematically determines which sublayers to offload to the CPU, and **(C2)** the execution back-end, which seamlessly integrates both the Intel CPU with AMX and the NVIDIA GPU for cooperative LLM inference acceleration. While prior work such as FlexGen [43] and FastDecode [23] *only* compute-offloads the least compute-intensive sublayer to the CPU, LIA allows all sublayers to be compute-offloaded to an AMX-enabled CPU based on batch size and sequence length. **(C1)** holistically considers several factors to determine optimal offloading policy, including the operations per byte of each sublayer for a given batch size and input token length, the volume of data transferred between the CPU and GPU for each sublayer, and the compute throughput and memory bandwidth of both CPU and GPU. By taking these variables into account, LIA maximizes resource utilization and minimizes end-to-end latency.

In particular, LIA leverages a unique characteristic of LLMs: *the fluctuating dynamic range of operations per byte across their sublayers for different batch sizes and input token lengths.* This enables the development of an offloading policy that minimizes end-to-end inference latency for a given batch size and input token length leveraging the heterogeneity of the system. **(C2)** extends the Intel Extension for PyTorch (IPEX), originally designed for CPU- or GPU-only acceleration, to enable seamless cooperative LLM inference acceleration across both the Intel CPU with AMX and the NVIDIA GPU. This back-end extension also introduces optimizations that further enhance the utilization of GPU memory and both CPU and GPU's compute resources. Deploying OPT-66B and OPT-175B on an SPR-H100 (GNR-H100) system, LIA demonstrates significant end-to-end performance improvements, delivering 4.0–7.0× (10–19×) lower latency and 1.2–3.7× (1.6–5.1×) higher throughput compared to the latest framework for latency-sensitive and throughput-driven LLM inference scenarios.

**Memory-offloading policy to use CXL for throughput-driven LLM inference acceleration (§6).** For throughput-driven applications [29, 33, 34], increasing batch size is essential to achieve high throughput. However, current DDR memory capacity, limited by the number of DDR channels, restricts the increase of batch size. For instance, increasing the batch size from one to 256 for an input sequence length of 1024 with OPT-175B raises the memory requirement from 330 GB to a staggering 1.6TB. To meet such substantial memory needs in a *cost-efficient* manner, we propose using affordable CXL memory, composed of DDR4 DRAM modules repurposed from retired data center servers [54], alongside DDR memory as a capacity expander. While CXL memory is cheaper and requires 3× fewer pins than DDR, it also has higher latency and lower bandwidth compared to DDR [48]. To mitigate these drawbacks and avoid compromising LLM inference throughput,

[1] Even with 4-bit quantization —which often sacrifices model quality— OPT-175B requires at least two H100 GPUs to store the model parameters alone, excluding intermediate values such as activations and KV-cache.

[2] Transferring the model parameters of OPT-175B between H100 and the CPU through PCIe 5.0 incurs an additional ∼5 seconds of inference latency.

[3] §4 includes additional microbenchmarking results for the earlier NVIDIA GPU generations, P100 and V100.

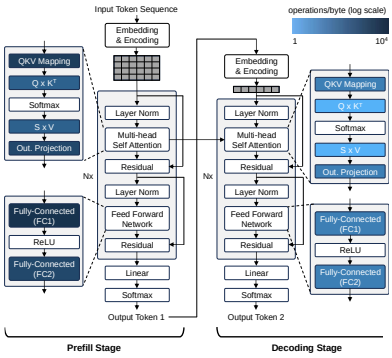


**Figure 1: OPT model architecture along with inference data flow, where the heatmap depicts the operations/byte of sublayers of the prefill and decoding stages for OPT-175B with an input token length $L$ of 512 and a batch size $B$ of 180.**

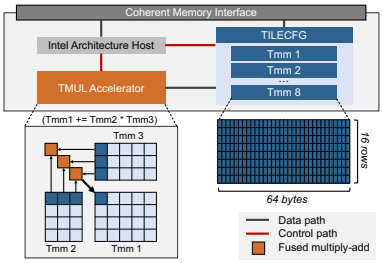


**Figure 2: Intel Advanced Matrix Extensions (AMX).**

we develop a synergetic memory-offloading policy that *stores all parameters in CXL memory while keeping intermediate values in DDR memory*. The policy is primarily designed to eliminate the cost of low-latency PCIe transfers between the CPU and CXL memory. By incorporating CXL memory, LIA reduces DDR memory usage by up to 43% for a given batch size. Alternatively, it can deliver up to 1.45× higher throughput by supporting larger batch sizes (up to 1.76×) within DDR memory capacity constraints.

# 2 Background

## 2.1 LLM Inference

An LLM typically consists of an embedding/encoding layer, $N$ decoder layers, and a language model (LM) head (linear and softmax). Among these layers, we focus on explaining the decoder layers since they collectively dominate the inference time and the memory consumption. A decoder layer comprises multiple sublayers consisting of matrix multiplication and other operations such as layer normalization, residual, and softmax. All $N$ decoder layers share an identical structure while each has their unique set of parameters. When receiving an input token sequence, a given model passes it through all the layers above to generate an output token (*i.e.,* prefill or Sum stage). $K$ and $V$ matrices corresponding to the input sequence are computed and stored on the memory as $KV$ cache during this stage. Subsequently, the model takes this generated output token as the new input, and passes it through its layers to produce the next output token (*i.e.,* decoding or Gen stage), which continues recursively until it generates a complete output token sequence. Fig. 1 depicts the network architecture of OPT-175B highlighting the prefill stage and the first decoding stage.

The layers of the decoding stage can be computed for a single input token or a batch of input tokens, with the exception of the attention scoring sublayers. For attention scoring ($Q \times K^\top$ and $S \times V$ sublayers), $KV$ vectors from the $QKV$ mapping sublayer are concatenated with the $KV$ cache, which are first generated by the prefill stage and repeatedly used for computing the attention scoring sublayers for subsequent input tokens. However, $KV$ cache are not shared across the batch since they are unique to each input token within the batch.

Lastly, depending on the stage, input token length ($L$), and batch size ($B$), the operations/byte across sublayers in a layer, as well as that of the same sublayer across layers can widely vary. For example, the operations/byte of sublayers in OPT-175B can range from 1 to 50,000 for $L = 512$ and $B = 180$, which is visually represented with a heat map in Figure 1. Note that softmax, layer normalization, and residual sublayers in the decoder layer are commonly fused with appropriate adjacent sublayers as they have low operations/byte.

## 2.2 Intel Advanced Matrix Extensions (AMX)

To significantly improve the throughput of the CPU for machine learning (ML) applications, Intel has integrated AMX, an on-chip matrix-multiplication accelerator, along with Instruction Set Architecture (ISA) support, starting from the 4th generation Xeon CPUs (SPR), released in 2023. Figure 2 depicts the accelerator architecture consisting of two core components: (1) a 2D array of registers (tiles) and (2) Tile matrix multiply unit (TMUL) [6], which are designed to support INT8 and BF16 formats. The tiles store sub-arrays of matrices and the TMUL, a 2D array of multiply-add units, operate on these tiles. The 2D structure of TMUL delivers significantly higher operations/cycle through large tile-based matrix computation, which operates with greater parallelism than 1D compute engines such as AVX engines. The CPU dispatches AMX instructions, such as tile load/store and accelerator commands, to the multi-cycle AMX units. The accelerator's memory accesses remain coherent with the CPU's memory accesses.

## 2.3 CXL Memory

It has become more expensive for the current DRAM and its (parallel) DDR interface technologies to meet the capacity and bandwidth demanded by datacenter servers under various physical constraints imposed by the CPU package and the PCB board. To address this challenge, CXL [42] has emerged as a promising memory interface that can cost-efficiently expand the capacity and bandwidth of a memory system, complementing the traditional DDR interface. For example, CXL built on the (serial) PCIe interface requires 3× fewer pins [48] compared to DDR5 DIMM slots. Furthermore, the CXL protocol can expose CXL memory as memory in a remote NUMA node while decoupling memory technologies from a specific memory interface of a CPU. Therefore, it can connect any past-generations DDR DRAM (*e.g.,* DDR4) to the CPU although

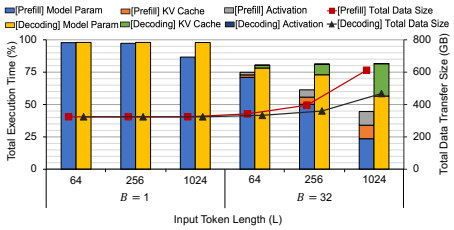


**Figure 3: Latency of OPT-175B prefill and decoding stages contributed by the CPU-GPU transfers of parameters, $KV$ cache, and activation, along with the CPU-GPU transfer amount.**

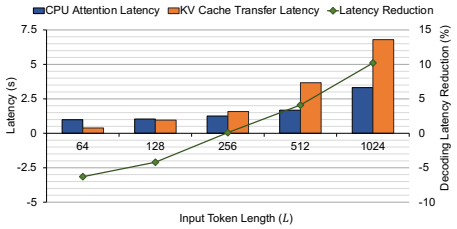


**Figure 4: Latency comparison between computing CPU-offloaded sublayer and transferring $KV$ cache to GPU, and latency reduction via compute-offloading at $B = 32$.**

the CPU does not support it anymore. Nonetheless, CXL increases the latency of memory accesses by 140–170 *ns* compared to DDR and supplements only up to 50% bandwidth of what DDR memory provides for a system, with multiple modules [48].

## 3 Performance Bottlenecks of Offloading Frameworks for LLM Inference

The latest frameworks facilitate LLM inference with a single GPU for models that exceed the GPU memory capacity [13, 22, 43]. Specifically, memory-offloading stores all parameters and intermediate values to CPU memory or even storage, and then transfers parameters and intermediate values required for the GPU to compute a given layer to GPU memory over PCIe. In addition to memory-offloading, compute-offloading has been proposed, directing the compute of attention scoring sublayers to CPU to reduce the CPU-GPU data transfer. In this section, we use FlexGen [43], the latest offloading framework for LLM inference, to run OPT-175B with various $L$ and $B$ values on an SPR-A100 system. See §7 for the system configuration. Data transfer time for model parameters, KV cache, and activation ( §3.1, §3.2) is measured using NVIDIA Nsight Systems [1] with NVTX markers [2], while CPU attention latency (§3.2) is measured using Python Time Module. For $B = 1$, all parameters are offloaded to CPU memory while the $KV$ cache and activation are stored in GPU memory. For $B = 32$, $KV$ cache and activation are also offloaded to CPU memory because they demand more memory capacity for larger $B$ and thus cannot fit into GPU memory. We analyze the performance bottlenecks of FlexGen and provide the following insights.

**Insight-1:** Memory-offloading presents long inference latency dominated by the slow CPU-GPU transfer.
**Insight-2:** Compute-offloading obviates the CPU-GPU transfer of $KV$ cache but increases inference latency when the CPU compute time gets longer than the CPU-GPU transfer time.

### 3.1 Memory-Offloading: Slow Data Transfer

Figure 3 shows that the CPU-GPU transfer of parameters contributes to more than 98% of the latency of computing the prefill and decoding stages for short $L$ when $B$ is 1. For long $L$, the percentage of the latency contributed by the CPU-GPU transfer during the prefill stage decreases to 87%. This is because the prefill stage performs more computation, proportional to $L$, with the same number of parameters transferred to GPU memory. However, the decoding stage conducts almost the same amount of computation regardless of $L$ for the same $B$, and thus the percentage remains the same.

When $B$ increases to 32, the size of $KV$ cache and activation, which increases with $B{\times}L$, can demand as large as 145GB. Therefore, $KV$ cache and activation should also be stored in CPU memory and then transferred to GPU memory on demand, because they do not fit into the GPU memory. Meanwhile, although the amount of computation also increases with $B{\times}L$, it does so at a faster rate than the amount of CPU-GPU transfer. For example, increasing $L$ from 64 to 1024, the amount of computation increases by 16.2×, whereas that of CPU-GPU transfer increases by only 1.8×. Hence, the percentage of the latency of computing the prefill stage contributed by the CPU-GPU transfer notably decreases with $L$. In contrast to the prefill stage, the percentage of the latency of computing the decoding stage contributed by the CPU-GPU transfer still remains above 80%, regardless of $L$. This is because the amount of computation for the decoding stage scales with $B$, not $B \times L$.

### 3.2 Compute-Offloading: Limited CPU Compute Throughput

$KV$ cache is first generated by the prefill stage and then continuously concatenated with $KV$ vectors generated during multiple decoding stages. To obviate the CPU-GPU transfer of the $KV$ cache during the decoding stage, FlexGen has proposed to offload the computation of the attention scoring sublayers, which uses the $KV$ cache, to the CPU. Figure 4 shows that the latency of computing the decoding stage reduced by compute-offloading the attention scoring sublayers to the CPU is only up to 10.2% at $L = 1024$. This is because of two reasons. (1) The CPU-GPU transfer of parameters is still responsible for a large percentage of the latency, and (2) the latency of the CPU computing the attention scoring sublayers is considerable, compared to the latency of CPU-GPU transfer of $KV$ cache (*e.g.,* 1 *s* vs. 0.4 *s* in Figure 4). Note that the measured maximum matrix-multiplication throughput of SPR with AVX512 is less than 1% of that of A100. This leads to a significant amount of time for the CPU to compute the attention scoring sublayers, even increasing the inference latency when $L$ is short ($L = 64$ and 128 in Figure 4).

## 4 Matrix-Multiplication Throughput of AMX

This section presents the matrix-multiplication throughput of AMX using microbenchmarks derived from OPT-175B. The throughput is measured on SPR (40-core) and GNR (128-core) CPUs and compared

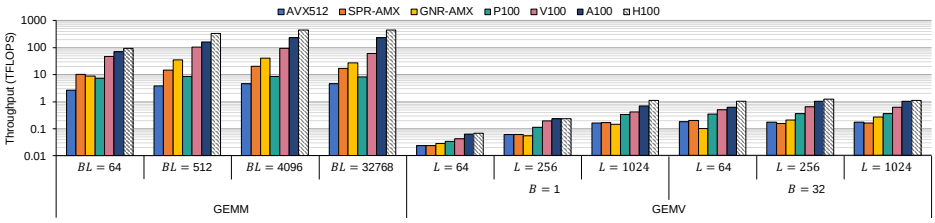


**Figure 5: GEMM and (batched) GEMV throughput of AVX512, SPR-AMX, GNR-AMX, P100, V100, A100, and H100.**

against AVX512 on SPR and four generations of Nvidia GPUs: P100, V100, A100, and H100. The evaluation uses GEMM and GEMV microbenchmarks to reflect real-world workloads in LLM inference. The GEMM benchmark simulates the compute-intensive FC1 sublayer of the prefill stage, involving matrix multiplication with dimensions $(B \times L, d_{model})$ and $(d_{model}, 4 \times d_{model})$, where $d_{model}$ is the model dimension (*e.g.,* 12,288 for OPT-175B). The GEMV benchmark captures the memory-bound $Q \times K^\top$ sublayer of the decoding stage, involving batched multiplication of a vector and a matrix with dimensions $(B \times n_h, 1, d_h)$ and $(B \times n_h, d_h, L)$, where $n_h$ and $d_h$ represent the number of heads and head dimensions, respectively. Benchmarks are conducted across a range of $B \times L$ values for GEMM and $B/L$ values for GEMV, using half-precision floating-point formats supported by each architecture: BF16 for AMX, A100, and H100, and FP16 for AVX512, P100, and V100. The FC1 and $Q \times K^\top$ sublayers are selected as they represent the extremes in operations per byte among all sublayers in OPT-175B —FC1 being the most compute-intensive and $Q \times K^\top$ the most memory-intensive. Our extensive evaluation offers the following key insight:

**Insight-3:** AMX delivers up to 4.4× (and potentially 8×) higher throughput than AVX and offer competitive performance against earlier generations of GPUs, making compute-offloading a far more attractive option compared to CPUs supporting only AVX.

## 4.1 GEMM Throughput: Compute-Bound

Figure 5 (left) shows GEMM throughput across different architectures. The general performance ranking from highest to lowest is: H100, A100, V100, GNR-AMX, SPR-AMX, P100, and AVX512. SPR-AMX achieves a maximum theoretical throughput of 90.1 TFLOPS, which is 8× and 4.7× higher than AVX512 and P100, respectively. Measured maximum throughput of SPR-AMX is 4.5× and 2.4× higher than AVX512 and P100, respectively, over the range of evaluated $B \times L$. SPR-AMX achieves lower utilization of peak performance as the recently-introduced AMX libraries are less optimized compared to mature libraries for AVX and P100.[4] When compared to recent GPUs, SPR-AMX achieves achieves 14–28%, 7–15%, and 4–11% of the throughput of V100, A100, and H100, respectively, at $B \times L$ = 64–36,684. Throughput scalability of SPR-AMX is limited by its 40-core count, as AMX performance proportionally scales with the number of cores. GNR-AMX addresses this with 128 cores, delivering up to 2.4× higher throughput than SPR-AMX. A two-socket GNR system further increases the throughput by 1.8×, achieving 68%, 30%, and 16% of the throughput of V100, A100, and H100, respectively. Despite lower raw throughput, AMX's large memory capacity enables competitive end-to-end performance by eliminating the need for slow CPU-GPU transfers, particularly for LLM inference tasks that exceed GPU memory capacities (§7).

[4]For well-optimized GEMM shapes, *e.g.,* (4K, 4K) × (4K, 4K), AMX achieves 7× higher throughput compared to AVX512.

## 4.2 GEMV Throughput: Memory-Bound

Figure 5 (right) illustrates the GEMV throughput across different architectures, with a slightly altered performance ranking: H100, A100, V100, P100, GNR-AMX, SPR-AMX, and AVX512. This change in the order attributes to varying ratio between compute and memory bandwidth in these systems. With 8 DDR5-4800 channels, SPR-AMX delivers a peak throughput of 199 GFLOPS, which is nearly identical to AVX512, differing by less than 10%. This similarity arises from the memory-bound nature of GEMV workloads and that both AMX and AVX512 share the same memory bandwidth (260 GB/s on SPR system). SPR-AMX achieves 54%, 31%, 19%, and 15% of the GEMV throughput of P100, V100, A100, and H100, respectively, consistent with their relative memory bandwidths of 41%, 34%, 20%, and 15%. At smaller matrix and vector sizes ($B$ and/or $L$), SPR-AMX achieves relatively higher throughput compared to GPUs, reaching 70%, 57%, 38%, and 35% of the throughput of P100, V100, A100, and H100, respectively. This improvement is due to the reduced overhead of GPU kernel invocation, which becomes significant for smaller computations. Lastly, GNR-AMX improves GEMV throughput by 70% over SPR-AMX by leveraging its 12 DDR5-5600 channels, effectively doubling the available memory bandwidth.

# 5 LIA: AMX-aware Framework for CPU-GPU Cooperative LLM Inference Acceleration

The latest compute-offloading framework, FlexGen, offloads only the attention scoring sublayer to the CPU. This choice, made *empirically*, results in only modest latency reductions due to the limited throughput of AVX on older CPU generations before SPR (§3). Leveraging the high throughput of modern CPUs with AMX, we propose LIA, a framework for synergistic CPU-GPU cooperative LLM inference acceleration that effectively utilizes CPU compute resources. This framework is seamlessly integrated with Intel IPEX [7]. Figure 6 provides an overview of LIA. The framework assumes that the CPU stores all model parameters and intermediate values in its large memory, while the GPU computes the model layer by layer in a manner similar to naïve data-offloading frameworks. LIA formulates an optimization problem to systematically determine which sublayers to offload to the CPU to minimize inference latency for given batch size ($B$) and input sequence length ($L$). This

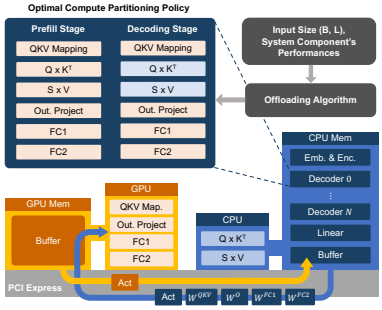


**Figure 6: Proposed CPU-GPU cooperative computing framework for LLM inference.**

optimization focuses on the decoding layers, which dominate LLM inference latency and memory usage (§2.1). The following subsections describe LIA's compute-offloading algorithm, performance optimizations techniques, and integration with Intel IPEX.

## 5.1 Compute-offloading Algorithm

The core of LIA's optimization lies in determining an offloading policy represented by a vector $\mathbf{p} = (p_1, p_2, \cdots, p_6)$, where each element $p_i \in \{0, 1\}$ indicates whether the $i^{th}$ sublayer of a decoder layer (with six sublayers as shown in Figure 6) is computed on CPU ($p_i = 1$) or GPU ($p_i = 0$). The optimal compute-offloading policy minimizes the latency (or maximize throughput) of LLM inference, experessed as:

$$T(\mathbf{p}_{\text{opt}}) = \min_{\mathbf{p}} T(\mathbf{p}), \tag{1}$$

where $T(\mathbf{p})$ represents the latency of computing a single decoder layer for the corresponding offloading vector and $\mathbf{p}_{\text{opt}}$ is the optimal offloading vector. The latency of a single decoder layer is broken down into three components for each sublayer:

$$T(\mathbf{p}) = \sum_{i=1}^{6} \left( T_{i,load}(\mathbf{p}) + T_{i,comp}(\mathbf{p}) + T_{i,store}(\mathbf{p}) \right), \tag{2}$$

where $T_{i,load}(\mathbf{p})$, $T_{i,comp}(\mathbf{p})$, and $T_{i,store}(\mathbf{p})$ represent the latencies for loading the two operand matrices to a given compute device, performing sublayer computation, and storing intermediate results to CPU, respectively.

First, the data load latency of sublayer $i$ is expressed as follows:

$$T_{i,load}(\mathbf{p}) = T_{i,load,X_i}(\mathbf{p}) + T_{i,load,Y_i}(\mathbf{p}) + T_{i,load,R_i}(\mathbf{p}), \tag{3}$$

where $T_{i,load,X_i}(\mathbf{p})$, $T_{i,load,Y_i}(\mathbf{p})$, and $T_{i,load,R_i}(\mathbf{p})$ accounts for the data movement of activation (also called hidden states), model parameters or KV matrices (or cache), and the activation of one of the previous sublayers for residual operation, respectively. $T_{i,load,X_i}(\mathbf{p})$, $T_{i,load,Y_i}(\mathbf{p})$, and $T_{i,load,R_i}(\mathbf{p})$ are obtained as follows:

**Table 1: Data size and compute count of GEMM/GEMV sublayers in a Decoder layer of OPT models for BF16 data type.**

| Stage | Sublayer | $D_X$ (Byte) | $D_Y$ (Byte) | $C$ (FLOP) |
|---|---|---|---|---|
| Prefill | QKV Mapping | $2BLd_m$ | $6d_m^2$ | $6BLd_m^2$ |
| | $Q \times K^\intercal$ | $2BLd_m$ | $2BLd_m$ | $2BL^2d_m$ |
| | S × V | $2BLd_m$ | $2BLd_m$ | $2BL^2d_m$ |
| | Out. Projection | $2BLd_m$ | $2d_m^2$ | $2BLd_m^2$ |
| | FC1 | $2BLd_m$ | $8d_m^2$ | $8BLd_m^2$ |
| | FC2 | $8BLd_m$ | $8d_m^2$ | $8BLd_m^2$ |
| Decoding | QKV Mapping | $2Bd_m$ | $6d_m^2$ | $6Bd_m^2$ |
| | $Q \times K^\intercal$ | $2Bd_m$ | $2BLd_m$ | $2BLd_m$ |
| | S × V | $2Bd_m$ | $2BLd_m$ | $2BLd_m$ |
| | Out. Projection | $2Bd_m$ | $2d_m^2$ | $2Bd_m^2$ |
| | FC1 | $2Bd_m$ | $8d_m^2$ | $8Bd_m^2$ |
| | FC2 | $8Bd_m$ | $8d_m^2$ | $8Bd_m^2$ |

$$T_{i,load,X_i}(\mathbf{p}) = \begin{cases} \frac{D_{X_i}}{BW_{PCIe}} & \text{if } p_i \oplus p_{i-1} = 1 \\ 0 & \text{else,} \end{cases} \tag{4}$$

$$T_{i,load,Y_i}(\mathbf{p}) = \begin{cases} \frac{D_{Y_i}}{BW_{PCIe}} & \text{if } p_i = 1 \\ 0 & \text{else,} \end{cases} \tag{5}$$

$$T_{i,load,R_i}(\mathbf{p}) = \begin{cases} \frac{D_{X_i}}{BW_{PCIe}} & \text{if } i = 4 \text{ and } p_i \oplus p_1 = 1 \\ \frac{D_{X_i}}{BW_{PCIe}} & \text{if } i = 6 \text{ and } p_i \oplus p_4 = 1 \\ 0 & \text{else,} \end{cases} \tag{6}$$

where $D_{X_i}$ and $D_{Y_i}$ denote the data size of the first and second matrix operand of sublayer $i$, respectively, and $BW_{\text{PCIe}}$ denotes the PCIe bandwidth. $D_{X_i}$ and $D_{Y_i}$ of each sublayer is summarized in Table 1. We set $p_0 = p_6$, as the activation for sublayer 1 will be placed on the device where the previous decoder layer's sublayer 6 was computed. Equation (5) does not hold for the cases of $i = 2, 3$ during the prefill stage, which in these cases

$$T_{i,load,Y_i}(\mathbf{p}) = \begin{cases} \frac{D_{Y_i}}{BW_{\text{PCIe}}} & \text{if } p_i \oplus p_1 = 1 \\ 0 & \text{else.} \end{cases} \tag{7}$$

This is because $Y_2$ (Key) and $Y_3$ (Value) during the prefill stage are generated from sublayer 1. Next, the computation latency is

$$T_{i,comp}(\mathbf{p}) = \begin{cases} \frac{D_{X_i}+D_{Y_i}}{BW_{\text{GPU}}} + \frac{C_i}{TH_{\text{GPU}}} & \text{if } p_i = 1 \\ \frac{D_{X_i}+D_{Y_i}}{BW_{\text{CPU}}} + \frac{C_i}{TH_{\text{AMX}}} & \text{if } p_i = 0, \end{cases} \tag{8}$$

where $BW_{\text{GPU}}$ and $BW_{\text{CPU}}$ are the memory bandwidths of GPU and CPU, respectively, $C_i$ is the computation count of sublayer $i$, and $TH_{\text{GPU}}$ and $TH_{\text{CPU}}$ are the BF16 compute throughput of GPU and CPU, respectively. Finally, the store latency is

$$T_{i,store}(\mathbf{p}) = \begin{cases} \frac{D_{KV}}{BW_{\text{PCIe}}} & \text{if } i = 1 \text{ and } p_i = 1 \\ 0 & \text{else,} \end{cases} \tag{9}$$

where $D_{KV}$ is the KV cache size generated in sublayer 1. Note that the amount of computation is a function of only $B$ and $L$, while the amount of CPU-GPU transfer is a function of not only $B$ and $L$ but also the offloading choice of adjacent sublayers.

## 5.2 Performance Optimization

To further reduce latency, we deploy two optimizations to better utilize to compute and memory resources of CPU and GPU.

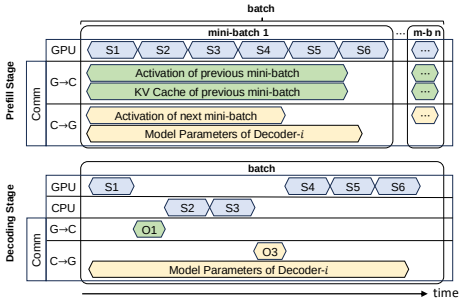


**Figure 7: Timing diagram of a single Decoder layer with overlapping. Example case of prefill policy p = (0, 0, 0, 0, 0, 0) and decoding policy p = (0, 1, 1, 0, 0, 0). S{i} and O{i} denote the computation and output data of sublayer *i*, respectively.**

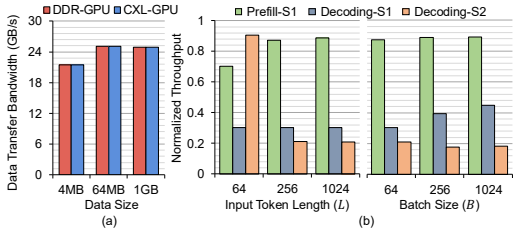


**Figure 8: (a) Bandwidth of DDR-GPU and CXL-GPU transfer across different data sizes. (b) throughput of the CPU getting model parameters and $KV$ cache from CXL memory, normalized to DDR memory.[5]**

**Optimization-1: Efficient use of GPU memory.** LIA begins by storing all parameters and intermediate values in CPU memory, transferring only the required data to the GPU for computing the current sublayer. This process often leaves a large portion of GPU memory unused, especially for small batch sizes where total capacity exceeds the needs of a single Decoder layer. For example, over 90% of A100 memory remains unused during OPT-30B inference with $B = 1$. While FlexGen utilizes this unused GPU memory to store as many sublayers of all decoder layers as possible, LIA takes a different approach. Instead, it stores all sublayers of as many decoder layers as possible in the unused GPU memory. This method allows LIA to make more efficient and granular use of GPU memory capacity, while also reducing the need for CPU-GPU data transfers. For example, in the case of OPT-30B, LIA and FlexGen increase GPU memory usage by approximately 1.2 GB per decoder layer and 4.7 GB per sublayer of all decoder layers, respectively. With $B = 1$ and $L = 2016$ (*i.e.,* maximum input token sequence length), LIA stores 62% of decoder layers using 35 GB of GPU memory, while FlexGen stores only 58% of sublayers with 32 GB on an A100 GPU.
**Optimization-2: Overlapping CPU/GPU computation with CPU-GPU transfer.** Overlapping computation and data transfer minimizes CPU and/or GPU idle time during transfers. Figure 7 illustrates LIA's overlapped execution timing diagram. During the prefill stage, LIA follows FlexGen's approach of splitting a batch into two or more mini-batches, overlapping PCIe transfers of model parameters of the next Decoder layer and intermediate values from/for adjacent mini-batches with the computation of the current mini-batch to reduce pipeline bubbles [43]. However, in the decoding stage, LIA overlaps computation of the entire batch with parameter transfers, unlike FlexGen, which uses mini-batch overlapping. Recent studies show [37, 51] that this approach negatively impacts decoding, as compute time does not scale linearly with smaller mini-batches, diminishing the benefit of reduced pipeline bubbles. By avoiding mini-batch overlap in decoding, LIA achieves a 1.1–1.3× lower latency than FlexGen for OPT-175B inference at $B = 900$.

## 5.3 Full-System Framework Implementation

We implement the full LIA framework by extending the latest IPEX library. The original IPEX library is built using `pytorch-cpu` and `pytorch-CXX` distributions for Intel CPUs and GPUs, which are incompatible with NVIDIA GPUs. Thus, we rebuilt the IPEX library, binding it to a `pytorch-cuda` distribution. IPEX Transformers [7] and HuggingFace Transformers [5] libraries are modified for LIA's compute-offloading algorithm and performance optimization techniques. First, LIA framework determines the optimal offloading policy based on the input characteristics. These include model parameters, intermediate values for each sublayer, batch size ($B$), and input length ($L$). The framework also considers system performance characteristics, such as the bandwidth of the CPU-GPU interface and the compute throughput and memory bandwidth of both the CPU and GPU. Leveraging this information to schedule computations and CPU-GPU data transfers. Finally, it executes the model using the derived optimal offloading policy, ensuring efficient and effective inference.

# 6 Exploiting CXL Memory for LLM Inference

As model sizes grow, even large CPU memory becomes insufficient and cost-inefficient for storing all parameters and intermediate values, particularly when batch size ($B$) and input length ($L$) are large. For instance, OPT-175B with $B = 1024$ and $L = 256$ requires approximately 1.4 TB of memory. To address this challenge, we advocate using CXL memory, composed of inexpensive DDR4 modules from retired datacenter servers [54], for offline LLM inference with large batch sizes ($> 860$). Despite its cost advantage, CXL memory has limitations: its bandwidth is typicall only 50% of DDR memory, and its latency is 2–3× higher [48]. To overcome these challenges, we propose a memory-offloading policy that stores all model parameters in CXL memory when $B$ is large, leveraging key observations about its bandwidth and impact on compute throughput.
**Observation-1: CXL does not hurt the bandwidth of CPU-GPU transfer of parameters.** While DDR memory subsystems offer higher bandwidth, the bandwidth of CPU-GPU transfers is ultimately constrained by the PCIe interface. Therefore, as far as the CXL memory can provide a higher bandwidth than the PCIe bandwidth between the CPU-GPU, CXL memory achieves the same bandwidth for CPU-GPU data transfers as DDR memory. When a single CXL memory expander provides lower bandwidth than the PCIe connection between the CPU and GPU, multiple CXL memory can be interleaved with page-granularity NUMA memory allocation.

[5] {Prefill/Decoding}-{S1/S2} denotes the computation of sublayer {1/2} during the {prefill/decoding} stage. $B$ and $L$ are fixed to 64 and 256 as we sweep $L$ and $B$, respectively.

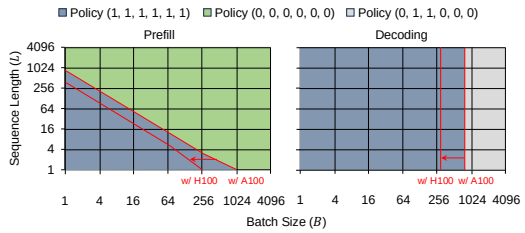


**Figure 9: Optimal compute offloading policies for OPT-175B across different combinations of $L_{in}$ and $B$ for SPR-A100 system and SPR-H100 system.**

This interleaving effectively scales the CXL-to-GPU data transfer bandwidth. Figure 8(a) illustrates that by interleaving two CXL memory expanders, each offering 17 GB/s of memory bandwidth, the combined bandwidth approaches the maximum achievable from DDR memory when transferring large data sizes (e.g., ≥ 300 MB per sublayer of the OPT-175B model). This parity ensures that storing parameters in CXL memory does not degrade GPU performance during data transfers.

**Observation-2: Obliviously storing all data in CXL memory leads to significant performance degradation.** While CXL memory does not impact GPU data transfer, storing the data in CXL memory significantly degrades CPU compute throughput. Figure 8(b) compares the throughput of the CPU AMX for sublayers 1 and 2 when parameters and $KV$ cache are stored in CXL memory, normalized to DDR memory. Sublayer 1 (multiplying activations with model parameters) experiences a throughput degradation of 11–70%, while sublayer 2 (multiplying activations with $KV$ cache in the decoding stage) suffers a greater reduction of 10–82%. Note that sublayer 2 does not involve model parameters as part of its computation. This disparity in degradation arises from the operations per byte for these sublayers. For sublayer 1, operations per byte scales with $B \times L$ in the prefill stage and $B$ in the decoding stage, whereas for sublayer 2, operations per byte remains constant at 1, regardless of $B$ or $L$. This difference makes sublayer 2 more sensitive to memory bandwidth and latency.

**Memory-offloading policy for throughput-driven inference.** For throughput-driven inference with large $B$, LIA's offloading policy derived from Equation (1) assigns all parameter-dependent sublayers (*i.e.,* sublayers 1, 4, 5, and 6) to GPU for both prefill and decoding stages. Based on Observation-1, the bandwidth parity ensures that GPU performance is unaffected by CXL offloading of model parameters. The $KV$ cache remains in the faster DDR memory in order to obviate the CPU compute performance degradation for $KV$-cache-dependent sublayers (*i.e.,* sublayers 2 and 3). As a result, even when all parameters are stored in CXL memory, LIA ensures that inference throughput remains uncompromised while optimizing memory costs.

## 7 Methodology and Evaluation

**System setup, benchmarks, and metrics.** We evaluate LIA using a Supermicro X13DDW-A server equipped with a 40-core Intel 4$^{th}$-generation Xeon Scalable Processor (SPR), paired with both A100 and H100 GPUs (Table 2). This setup enables us to assess two configurations: SPR-A100 and SPR-H100. For benchmarks, we evaluate BF16-based OPT-30B and OPT-175B on the SPR-A100 configuration, and OPT-66B and OPT-175B on the SPR-H100 configuration, as these models do *not* fit within a single instance of the GPUs. We compare LIA with two baselines: IPEX [7], the CPU-only framework which exploits AMX, and FlexGen [43], the latest offloading framework using GPU and AVX. We evaluate performance in two scenarios: (I) online inference, which is latency-driven and measured in seconds per query (s/query) with a batch size of $B = 1$; and (II) offline inference, which focuses on throughput and is measured in tokens per second (tokens/s) for batch sizes $B = 64$ and $B = 900$, covering diverse policy regions as shown in Figure 9. These batch sizes span the typical range evaluated in prior work on single-GPU inference [43].

**Table 2: Evaluation System.**

| **Supermicro X13DDW-A Server** |
|---|
| Intel® Xeon® Platinum 8460H Processor, 40 cores<br>8 DDR5-4800 channels, 512 GB DRAM |
| NVIDIA Tesla A100 Graphics Card, 40 GB HBM2, PCIe 4.0 |
| NVIDIA Tesla H100 Graphics Card, 80 GB HBM3, PCIe 5.0 |
| Samsung CXL Type-3 Memory Expander, 128 GB x2 |

**Token sequence lengths.** We determine representative input and output token sequence lengths, based on Azure LLM inference trace statistics [38]. Since the input token lengths are uniformly distributed, we set the input token length $L_{in}$ from 32 to maximum model-defined length of 2048, while output token lengths ($L_{out}$) are set to 32 and 256, corresponding to average lengths observed in code and conversation traces, respectively.

**Memory constraints and latency model.** Our evaluation system featuring 512GB of DDR memory cannot evaluate some combinations of $B$, $L_{in}$, and $L_{out}$ that require larger memory capacity. Although SPR supports up to 4 TB memory capacity per socket with 16×256 GB DIMMs, the cost per GB of such a high capacity DIMM is considerably more expensive than that of low capacity DIMMs. For example, the cost per GB of 256GB DIMMs is at least 2× more expensive than that of 32GB DIMMs [4]. To address these constraints, we use a latency model for evaluations beyond our system's capacity. The latency model computes the latency of a single Decoder layer for the prefill and decoding stages separately (Equation (2)) with the performance values evaluated in §7.2. It then sums the latencies of the two stages and multiplies by the number of decoder layers. This approach is analogous to prior works [41, 43] for performance estimation. The model exhibits an average error of 12% across measured points. Latency results derived from this analytical model are marked with stars (★) in the figures.

**Energy efficiency.** We compare the energy efficiency (energy/token) of LIA, IPEX, and FlexGen by measuring the system's average power consumption during inference using ipmitool [3]. Energy consumption per token is calculated by multiplying the measured power with inference latency and dividing by $L_{out}$.

### 7.1 Optimal Policies for Compute Offloading

Figure 9 shows the optimal compute-offloading policies for OPT-175B in the evaluation system (Table 2). For the prefill stage, the choice of offloading policy depends on the product of batch size

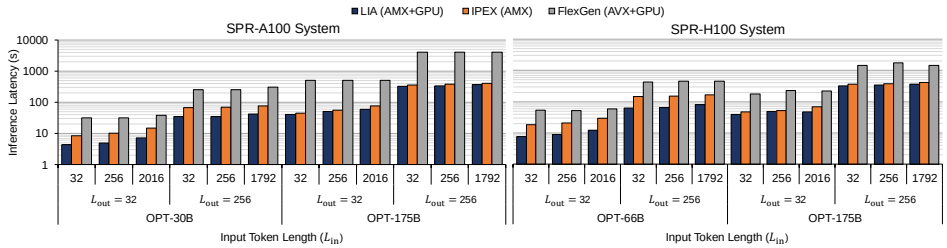


**Figure 10: Inference latency comparison among LIA, IPEX, and FlexGen for OPT-30B (only SPR-A100), OPT-66B (only SPR-H100), OPT-175B (both SPR-A100 and SPR-H100) across common input and output token sequence lengths.**

($B$) and input token length ($L$). When both $B$ and $L$ are small, all sublayers are offloaded to the CPU to minimize latency. As $B$ and $L$ increase, the GPU computes all sublayers to leverage its higher throughput. The transition between these two regions is approximately defined by $BL \simeq 850$ for OPT-175B. For the decoding stage, the choice of offloading policy is determined by $B$, independent of $L$. Thus, the offloading policy of LIA does not change during the decoding stage as the output tokens are generated. When $B$ is less than 858 for OPT-175B, all sublayers are offloaded to the CPU. As $B$ exceeds this threshold, the GPU computes the $QKV$ mapping, output projection, FC1, and FC2 sublayers, while the CPU continues to handle other sublayers.

**Policy selection across models.** LIA identifies three primary policies for compute offloading across all evaluated OPT models (OPT-30B, OPT-66B, and OPT-175B):

- Partial CPU Offloading $\mapsto \mathbf{p} = (0, 1, 1, 0, 0, 0)$
- Full CPU Offloading $\mapsto \mathbf{p} = (1, 1, 1, 1, 1, 1)$
- Full GPU Compute $\mapsto \mathbf{p} = (0, 0, 0, 0, 0, 0)$

However, the choice among the three policies can be different across models for given B and L because even the same sublayer for the same B and L demands different operations/byte and amounts of CPU-GPU transfer across them with different model sizes.

**Adaptability to other models.** The identified policies remain consistent for LLMs with architectures similar to the OPT family. However, for models like Mixture of Experts (MoE) [17], the diversity of offloading policies increases. For example, in MoE architectures, sublayers 5 and 6 exhibit reduced operations per byte as the number of experts increases, while their parameter sizes grow. Consequently, LIA may prefer policies such as $\mathbf{p} = (0, 1, 1, 0, 1, 1)$ instead of $(0, 1, 1, 0, 0, 0)$, as CPU computation for these sublayers becomes more efficient than CPU-GPU data transfer.

**Impact of GPU capability.** The GPU model significantly influences policy selection. H100 offers greater throughput, memory capacity, and bandwidth compared to A100. As a result, LIA more frequently selects GPU-centric policies [$\mathbf{p} = (0, 1, 1, 0, 0, 0)$ and $\mathbf{p} = (0, 0, 0, 0, 0, 0)$] for a broader range of $B$ and $L$ combinations when using H100. Despite this, LIA remains effective even with H100 systems, as it still opts for CPU-centric policy [$\mathbf{p} = (1, 1, 1, 1, 1, 1)$] for a significant portion of configurations.

## 7.2 Online Inference Latency

**SPR-A100.** Figure 10 (left) presents the inference latency of OPT-30B and OPT-175B using LIA, IPEX, and FlexGen on the SPR-A100 system. Across all combinations of $L_{in}$, $L_{out}$, and models, LIA consistently outperforms both baselines. Specifically, LIA achieves 1.8–2.1× (1.1–1.3×) and 5.3–7.3× (8.5–12×) lower latency compared to IPEX and FlexGen, respectively, for OPT-30B (OPT-175B). The primary advantage of LIA over FlexGen lies in significant reduction of CPU-GPU data transfer, the main bottleneck in FlexGen. Transfer reduction ranges from 31× to as much as 222,524× (§3.1). In contrast, LIA's performance gain over IPEX stems from two factors: first, leveraging unused GPU memory to store and compute decoder layers on the GPU, and second, deploying an all-GPU prefill policy (0, 0, 0, 0, 0, 0) for long $L_{in}$ values, as the prefill stage becomes more compute-intensive. The latency reduction trend remains consistent between $L_{out} = 32$ and 256, but the impact of leveraging GPUs during the prefill stage diminishes with larger $L_{out}$ as the decoding stage dominates the overall latency. With larger models like OPT-175B, the gap between LIA and IPEX reduces, as fewer decoder layers are stored on the GPU. Conversely, the gap between LIA and FlexGen increases, as LIA's relative CPU-GPU transfer amount over FlexGen decreases by up to 6.5× from OPT-30B to OPT-175B.

**SPR-H100.** Figure 10 (right) presents the same evaluation on the SPR-H100 system. For OPT-66B (OPT-175B), LIA achieves 2.1–2.5× (1.1–1.5×) and 4.9–7.0× (4.0–5.1×) lower latency compared to IPEX and FlexGen, respectively. The improved GPU performance, memory bandwidth, capacity, and PCIe bandwidth of the H100 contribute to a greater relative advantage over IPEX than on the A100 system. However, LIA's relative performance advantage over FlexGen decreases, as FlexGen benefits more from the increased GPU and PCIe performance. For OPT-175B, LIA on SPR-H100 achieves 1.1–1.3× lower latency than on SPR-A100 due to higher HBM3 memory bandwidth and PCIe 5.0.

## 7.3 Offline Inference Throughput

**SPR-A100.** Figure 11 (left) shows the offline inference throughput (tokens/s) for OPT-30B and OPT-175B using LIA, IPEX, and FlexGen on the SPR-A100 system. For OPT-30B (OPT-175B), LIA delivers 1.5–6.0× (1.1–6.1×) and 2.0–5.9× (1.3–6.0×) higher throughput compared to IPEX and FlexGen, respectively. For $B = 64$, LIA computes the prefill stage on the GPU and offloads the decoding stage to the CPU, avoiding CPU-GPU transfer overhead. As $L_{in}$ grows, the prefill stage becomes more compute-intensive, further increasing LIA's throughput advantage over IPEX. For instance, for OPT-30B with $L_{in} = 2016$ and $L_{out} = 32$, IPEX spends 92% of its time on the prefill stage, while LIA reduces prefill latency by 7.8× by leveraging

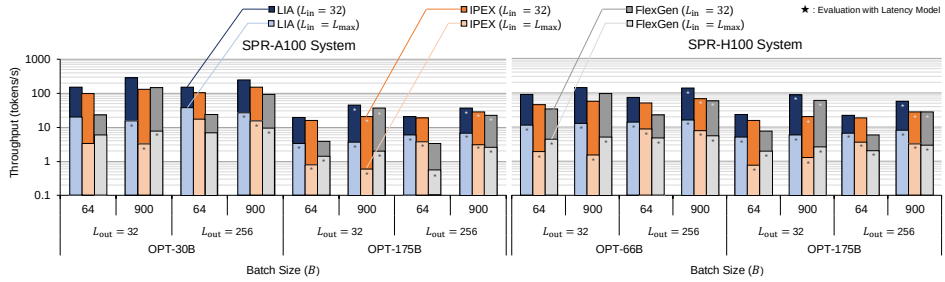


**Figure 11: Inference throughput (tokens/s) comparison between LIA, IPEX, and FlexGen for OPT-30B (only SPR-A100), OPT-66B (only SPR-H100), and OPT-175B models (both SPR-A100 and SPR-H100). $L_{max}$ is 2016 for $L_{out}$ = 32 and 1792 for $L_{out}$ = 256 and the bars labeled with stars are evaluated with latency model, while the unlabled bars are evaluated with the evaluation system.**

**Table 3: OPT-30B inference throughput of LIA with and without parameter-offloading to CXL for $B$ = 900.**

| $L_{in}$ | $L_{out}$ | Throughput (tokens/s) | | Offloaded Percentage |
|---|---|---|---|---|
| | | LIA | LIA w/ CXL[6] | |
| 32 | 32 | 280.38 | 280.50 (406.88) | 43.1% (30.1%) |
| | 64 | 293.50 | 292.70 (386.19) | 33.5% (25.2%) |
| | 128 | 282.86 | 284.93 (297.93) | 23.2% (19.1%) |
| | 256 | 233.36 | 235.70 (242.95) | 14.4% (12.6%) |

GPU computation. For $B$ = 900, LIA adopts the same offloading policy as FlexGen but achieves higher throughput by leveraging the higher compute throughput of AMX. Across different model sizes and $L_{out}$, the general trends remain similar. Figure 11 also highlights a significant throughput improvement as $B$ increases from 64 to 900 for LIA and FlexGen. However, this comes with increased memory requirements. Table 3 demonstrates that LIA reduces the DDR memory usage by up to 43.1% through the proposed CXL offloading while achieving within 1% of its performance without CXL. Under the same DDR memory footprint, CXL offloading enables increased $B$ values, achieving up to 1.45× higher throughput.

**SPR-H100.** On the SPR-H100 system, LIA delivers 1.3–8.3× (1.2–10×) and 1.2–3.3× (1.5–3.7×) higher throughput compared to IPEX and FlexGen for OPT-66B (OPT-175B). The throughput improvement of LIA over IPEX is higher than on the SPR-A100 system, enjoying the H100's greater GPU performance and PCIe bandwidth. However, LIA's relative performance over FlexGen is smaller compared to that on the SPR-A100 system, as FlexGen benefits more from these hardware enhancements. For the same OPT-175B, LIA achieves 1.1–1.9× higher throughput on SPR-H100 compared to SPR-A100. The improvement is greater for $B$ = 900, where the offloading policy utilizes GPU more aggressively vs. $B$ = 64.

## 7.4 Ablation Study and Breakdown

**Ablation study.** Table 4 illustrates the individual contribution of LIA's optimization techniques and compute-offloading policy. Optimization-1 significantly reduces latency for $B$ = 1, though the impact diminishes as $B$ increases. This is because the unused GPU memory decreases with larger $B$, resulting in fewer Decoder layers being stored on the GPU with optimization-1. In constrast, the benefit of optimization-2 improves as $B$ increases. For smaller $B$, the imbalance between the GPU-CPU transfer time and compute time results in limited overlap, while the data transfer time is more effectively overlapped with compute when $B$ = 900. LIA's compute-offloading policy delivers significant improvements of 6.2× and 3.5× over FlexGen's policy at $B$ = 1 and $B$ = 64, respectively, as LIA optimally utilizes the AMX-enabled CPU compute capability. At $B$ = 900, despite LIA deploying the same policy as FlexGen, it still improves over FlexGen by 1.9× by leveraging the enhanced AMX capability.

[6]CXL-offloading allows larger $B$ under the same DDR memory footprint as that of LIA without CXL-offloading. The value inside the parentheses is the throughput with increased $B$. The increased $B$ value is 1580, 1350, 1150, and 1050 for $L_{out}$ of 32, 64, 128, and 256, respectively.

**Table 4: Ablation study of optimization techniques and LIA's offloading policy. OPT-30B inference latency (s) for $L_{in}$ = 256 and $L_{out}$ = 32 on an SPR-A100 system.**

| Ablation Setting | $B$ = 1 | $B$ = 64 | $B$ = 900 |
|---|---|---|---|
| All optimizations | 5.05 | 24.00 | 291.02 |
| No Optimization-1 | 10.09 | 26.97 | 297.13 |
| No Optimization-2 | 5.05 | 26.96 | 443.61 |
| w/ FlexGen's policy | 31.07 | 84.80 | 291.02 |

**Table 5: Latency breakdown of LIA, IPEX, and FlexGen during OPT-30B inference for $L_{in}$ = 256 and $L_{out}$ = 32 on an SPR-A100 system. CPU, GPU, Com. denote for the CPU and GPU compute time, data transfer latency, respectively.**

| Batch Size | LIA | | | IPEX | FlexGen | | |
|---|---|---|---|---|---|---|---|
| | CPU | GPU | Com. | CPU | CPU | GPU | Com. |
| $B$ = 1 | 3.8 | 1.2 | 0.1 | 10.2 | 0.05 | 1.3 | 31.3 |
| $B$ = 64 | 16.9 | 7.7 | 3.9 | 75.7 | 20.9 | 9.8 | 86.0 |
| $B$ = 900 | 168.8 | 110.8 | 118.8 | 1216.5 | 505.0 | 98.7 | 128.7 |

**Runtime breakdown.** Table 5 presents the runtime breakdown of LIA, IPEX, and FlexGen during the OPT-30B inference on a SPR-A100 system, with overlap disabled. GPU compute time and data transfer latency is measured with Nsight Systems while CPU compute time is measured with Python Time Module. LIA improves over FlexGen by significantly reducing communication overhead and leveraging fast compute with AMX, while improving over IPEX by leveraging GPU to significantly reduce the total compute time.

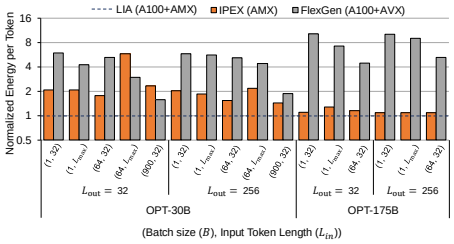


**Figure 12: Energy per token of LIA, IPEX, and FlexGen on an SPR-A100 system normalized to that of LIA. $L_{max}$ is 2016 for $L_{out}$ = 32 and 1792 for $L_{out}$ = 256.**

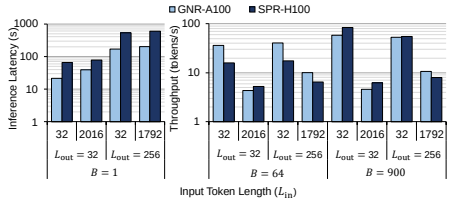


**Figure 13: OPT-175B online inference latency and offline inference throughput comparison between LIA on a GNR-A100 system and an SPR-H100 system.**

## 7.5 Energy Efficiency

Figure 12 shows the per-token energy consumption of IPEX and FlexGen normalized to that of LIA for various combinations of $B$, $L_{in}$, $L_{out}$, and OPT models on the SRR-A100 system. LIA achieves 1.1–5.8× and 1.6–10.3× higher energy efficiency compared to IPEX and FlexGen, respectively. For small $B$ ($\leq$ 64) and $L_{in}$ (= 32), LIA consumes significantly less energy than FlexGen due to its shorter inference latency, which reduces the impact of the system's static power on total energy consumption. As $B$ increases to 900, the latency gap between LIA and FlexGen narrows, leading to a reduction in the energy efficiency benefit to 1.6×. Conversely, LIA's energy benefit over IPEX grows with larger $B$ or longer $L_{in}$ (e.g., $B$ = 64, $L_{in}$ = $L_{max}$) as it leverages the GPU to handle compute-intensive tasks more energy-efficiently. Comparing A100 and AMX energy efficiency reveals that A100 is more efficient for compute-intensive tasks, as seen in specific measurements (e.g., $B$ = 64, $L_{in}$ = $L_{max}$ and $B$ = 900, $L_{in}$ = 32 for OPT-30B with $L_{out}$ = 32). LIA capitalizes on this by offloading such tasks to the GPU, further reducing energy consumption relative to IPEX. In addition, LIA reduces static energy consumption due to shorter inference latencies, a notable advantage over IPEX.

## 7.6 Scaling with Granite Rapids (GNR) CPUs

Table 6 summarizes the performance improvements relative to IPEX and FlexGen on GNR-A100 and GNR-H100 systems, featuring GNR CPUs with 128 cores. The performance gap between LIA and IPEX decreases by 14% on average when upgrading the CPU from SPR to GNR CPUs, attributed to IPEX's exclusive reliance on CPU performance, which benefits directly from GNR's enhanced compute throughput. In contrast, the performance gap between LIA and FlexGen increases by 1.7× on average, as LIA's heavier reliance on CPU compute gains more from GNR's improved capabilities.

**Table 6: Performance improvement of LIA over IPEX and FlexGen on GNR-A100 and GNR-H100 systems.**

| Inference Scenarios | Performance relative to | GNR-A100 | | GNR-H100 | |
|---|---|---|---|---|---|
| | | OPT-30B | OPT-175B | OPT-66B | OPT-175B |
| Online | IPEX | 1.5–1.7× | 1.1–1.2× | 1.5–1.8× | 1.2–1.4× |
| | FlexGen | 5.6–9.1× | 13–24× | 3.9–5.9× | 8.3–12× |
| Offline | IPEX | 1.1–4.2× | 1.1–4.1× | 1.3–3.6× | 1.1–4.4× |
| | FlexGen | 1.6–7.5× | 1.5–9.4× | 1.8–3.5× | 1.3–4.1× |

**CPU vs. GPU scaling.** To explore the impact of scaling CPU versus GPU capabilities, we compare LIA's performance on a GNR-A100 system to that on an SPR-H100 system. Figure 13 shows that for online inference, GNR-A100 achieves 1.4–2.0× lower latency compared to SPR-H100. For offline inference, GNR-A100 outperforms SPR-H100 for $B$ = 64 with up to 1.9× higher throughput but slightly lags for $B$ = 900, achieving 70% of SPR-H100's throughput on average. With 1.7× and 1.6× lower system cost ($) and higher energy-efficiency (tokens/s/W(TDP)), respectively, GNR-A100 delivers a more cost-efficient solution for LIA. This highlights the importance of choosing the right GPU-CPU combination to achieve better cost-efficiency.

## 7.7 Model Generalizability

LIA's optimizations are not limited to OPT models; they generalize across a wide range of LLMs sharing the same neural architecture backbone, decoder-only Transformer model with multi-head attention, such as GPT, Llama2, and Gemini. These models exhibit significant variability in operations/byte across stages and sublayers [24, 28, 36, 49, 49], which LIA effectively leverages on. Evaluation using the validated analytical model with SPR-A100/H100 and GNR-A100/H100 systems (described in §7), LIA consistently delivers 6.1–8.4× (1.4–7.6×), 7.4–10× (1.3–5.0×), and 7.6–11× (1.2–4.9×) lower latency (higher throughput) compared to FlexGen for Llama2-70B, Chinchilla-70B, and Bloom-176B models, respectively. Compared to IPEX, LIA achieves 1.2–1.4× (1.4–7.6×), 1.3–1.7× (1.3–4.8×), and 1.1–1.4× (1.1–6.5×) lower latency (higher throughput) for Llama2-70B, Chinchilla-70B, and Bloom-176B models, respectively.

## 7.8 Cost-efficiency vs. Multi-GPU System

Figure 14 compares per-GPU throughput and cost per million tokens[7] of LIA on a GNR-A100 system to multi-GPU inference on a DGX-A100 system, which deploys 8×A100 80GB GPUs connected via NVLink with 8-way Tensor Parallelism. DGX-A100 latency is evaluated using Vidur [12], an LLM inference system simulator developed by Microsoft. For $B$ = 1, LIA achieves 1.4–1.8× higher per-GPU throughput and 1.5–2.0× lower cost than multi-GPU inference. For $B$ = 64, LIA sees 30–33% lower per-GPU throughput and 1.3–1.4× higher cost than multi-GPU inference. However, assuming AMX reaches 50% of theoretical performance with improved

[7]The system cost of $22,000 for GNR-A100 and $200,000 for DGX-A100 is assumed to be amortized across the time period of 3 years and the power consumption is estimated with each system's TDP. Electricity cost of $0.1/kWh is assumed, the price in Lousiana, the cheapest in the U.S.

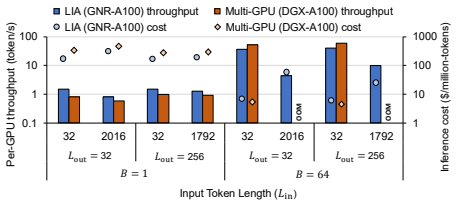


**Figure 14: Per-GPU throughput and cost comparison between LIA on a GNR-A100 system and multi-gpu inference on a DGX-A100 system for OPT-175B inference. OOM denotes for Out of Memory.**

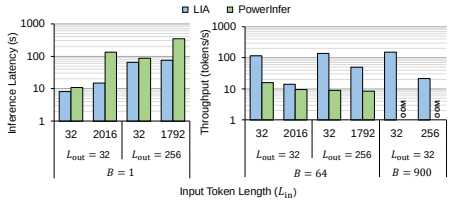


**Figure 15: Llama2-70B online inference latency and offline inference throughput comparison between LIA and PowerInfer on a GNR-A100 system.**

library optimization (as evaluated by our analytical model), LIA could achieve 1.1–1.3× higher per-GPU throughput and 1.1–1.2× lower cost even at $B = 64$. Comparison for $B = 900$ is omitted due to insufficient memory on the DGX-A100 system, while LIA supports this batch size and achieves higher throughput. Lastly, note that LIA requires only 10% of the system cost of GNR-A100 compared to multi-GPU inference on a DGX-A100 system, making it more accessible to a wider range of users.

## 7.9 Comparison to PowerInfer

Figure 15 shows that LIA achieves 1.4–9.0× lower inference latency and 1.5–15× higher inference throughput compared to PowerInfer [45] for Llama2-70B on a GNR-A100 system. PowerInfer runs into CUDA Out of Memory (OOM) for throughput-oriented large-batch ($B = 900$) scenarios, achieving limited benefit from large-scale batch processing. Optimized for systems with less competent CPU capability, PowerInfer requires frequent data transfer over PCIe for intra-layer communication between hot neurons on GPU and cold neurons on CPU. Such approach results in lower performance than LIA on a system equipped with AMX-enabled CPU even with model adaptation. Note that PowerInfer also focuses only on LLMs with high sparsity and suffers from accuracy loss.

# 8 Discussion

**Scaling to multi-GPU.** While this paper focuses on a single-GPU set up, LIA can be extended to multi-GPU systems facing memory capacity limits. When LIA directs a sublayer to the GPU, Tensor Parallelism can distribute computation across multiple GPUs. As GPU compute throughput and CPU-GPU bandwidth scale with GPU count, GPUs will handle computation more frequently than in a single-GPU setup. However, communication overhead between the GPUs may reduce the scaling impact, especially when the GPUs are connected via PCIe interconnects.

**The case of Grace-Hopper systems.** NVIDIA's Grace-Hopper system supports 900GB/s of CPU-GPU bandwidth, 7× higher than ×16 PCIe-5.0, with low CPU compute throughput[8]. The optimal policy derived from LIA in this system chooses all sublayers to be computed on the GPU since the CPU-to-GPU bandwidth is only 10% smaller than that of the CPU fetching the data. Based on evaluation with analytical model, the performance of LIA on a Grace-Hopper system achieves 1.8–2.3× lower latency and 3.0–4.1× higher throughput than a GNR-H100 system. Although LIA's compute-offloading strategy proves beneficial in bandwidth-constrained CPU-GPU systems, these findings suggest that improving CPU-GPU bandwidth may be a more effective direction than increasing CPU compute power for CPU-GPU collaborative computing, given the current CPU/GPU capability regime.

**Cost saving via CXL offloading.** CXL data offloading combined with AMX-enabled CPU compute offloading further reduces the system cost by allowing a portion of DDR memory to be replaced with cheaper CXL memory. Up to 43% of data required during inference can be offloaded to the CXL memory without compromising the inference throughput as presented in §7.3. For an OPT-175B model, this translates to a memory system cost reduction from $6,300 to $3,200[9], 8% and 9% reduction in total system cost of SPR-A100 and GNR-A100 configuration, respectively.

**Comparison to other cost-efficient systems.** LLM inference with only data-offloading using multiple cheaper GPUs (*e.g.,* P100 or V100) paired with a low-end CPU can be considered an alternative to LIA deploying a single high-end GPU (*e.g.,* A100 or H100) paired with an AMX-enabled CPU. For example, 3 V100 GPUs paired with a low-end CPU will have a similar cost to a GNR-A100 system. Our analysis with the analytical model shows that LIA deployed on a GNR-A100 system outperforms such system, delivering 6.3–11× longer latency and 2.2–16× lower throughput even ignoring inter-V100 communication overhead.

# 9 Related Work

**LLM inference with data offloading.** [13, 22] enable running LLMs larger than the accelerator memory capacity through CPU parameter offloading or storage devices. However, the batch size can be limited to a small value as they store *KV* cache and activation on the accelerator memory. FlexGen [43] offloads KV cache and activation to the CPU and proposes optimal data partitioning between GPU and CPU that minimizes the inference latency.

**CPU-GPU computing for LLM inference.** To obviate the KV cache data movement over the PCIe, FlexGen [43] and FastDecode [23] offloads attention scoring computation to CPU. However, the role of CPU is limited to only the attention scoring sublayer, the least compute-intensive sublayer, due to the low compute throughput of CPU. PowerInfer [45] partitions model parameters by processing hot neurons on GPU and others on the CPU. However, it

[8]The theoretical compute throughput of Grace CPU with Scalable Vector Extension version 2 (SVE2) instruction set is 6.91 TFLOPS, 30× lower than GNR, and the maximum memory bandwidth is 512 GB/s.

[9]A memory system configured with only DDR memory costs $11.25 per GB, while DDR and CXL each comprising half costs $5.60 per GB [10]

requires high sparsity to reduce CPU compute time and modifies the inference algorithm, limiting generalization across models.

## 10 Conclusion

We propose a single-GPU CPU-GPU cooperative LLM acceleration framework using Intel AMX and CXL memory. We show that AMX achieves competitive throughput with some GPUs. By developing a compute-offloading algorithm and implementing it on real hardware, LIA achieves up to 2.1× and 24× lower latency, 10× and 6.0× higher throughput, and 5.8× and 10.3× higher energy efficiency over Intel's AMX-based and the latest CPU-GPU collaborative LLM inference frameworks, respectively.

## Acknowledgments

This work was supported in part by grants from the IBM-Illinois Discovery Accelerator Institute (IIDAI) and from PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, by the MSIT, Korea, under the Global Scholars Invitation Program (RS-2024-00456287) supervised by the IITP, and by generous gifts from Intel Corp and Google.

## References

[1] 2025. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.
[2] 2025. NVIDIANVIDIA Nsight VSE Documentation. https://docs.nvidia.com/nsight-visual-studio-edition/2020.1/nvtx/index.html.
[3] Accessed in 2024. An open-source tool for controlling IPMI-enabled systems. https://github.com/ipmitool/ipmitool.
[4] Accessed in 2024. August 2024 Server Memory Prices. https://memory.net/memory-prices/.
[5] Accessed in 2024. HuggingFace Transformers Library. https://github.com/huggingface/transformers.
[6] Accessed in 2024. Intel® Architecture Instruction Set Extensions and Future Features. https://www.intel.com/content/www/us/en/content-details/774990/intel-architecture-instruction-set-extensions-programming-reference.html.
[7] Accessed in 2024. Intel® Extension for PyTorch. https://github.com/intel/intel-extension-for-pytorch.
[8] Accessed in 2024. Technical Overview Of The 4th Gen Intel® Xeon® Scalable processor family. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.htmll.
[9] Accessed in 2024. The Intel® Xeon® 6 Processor Family. https://www.intel.com/content/www/us/en/products/details/processors/xeon/xeon6-product-brief.html.
[10] Accessed in 2025. Memverge: CXL use case. https://memverge.com/cxl-use-case-slash-memory-costs-and-expand-capacity/.
[11] Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. 2024. On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
[12] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A Large-Scale Simulation Framework For LLM Inference. *Proceedings of Machine Learning and Systems* 6 (2024), 351–366.
[13] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–15.
[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2023. PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.
[15] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. GPT3.int8(): 8-bit Matrix Multiplication for Transformers at Scale. *Advances in Neural Information Processing Systems* 35 (2022), 30318–30332.
[16] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).
[17] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.
[18] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot. In *Proceedings of the International Conference on Machine Learning (ICML)*. 10323–10337.
[19] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. *arXiv preprint arXiv:2210.17323* (2023).
[20] Gemini Team. 2024. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805* (2024).
[21] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. 2023. Knowledge distillation of large language models. *arXiv preprint arXiv:2306.08543* (2023).
[22] Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun, and Benjamin Bossan. 2022. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate.
[23] Jiaao He and Jidong Zhai. 2024. FastDecode: High-Throughput GPU-Efficient LLM Serving using Heterogeneous Pipelines. *arXiv preprint arXiv:2403.11421* (2024).
[24] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556* (2022).
[25] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling Step-by-Step! Outperforming Larger Language Models with Less Training Data and Smaller Model Sizes. *arXiv preprint arXiv:2305.02301* (2023).
[26] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361* (2020).
[27] Dan Kondratyuk, Lijun Yu, Xiuye Gu, José Lezama, Jonathan Huang, Grant Schindler, Rachel Hornung, Vighnesh Birodkar, Jimmy Yan, Ming-Chang Chiu, et al. 2023. Videopoet: A large language model for zero-shot video generation. *arXiv preprint arXiv:2312.14125* (2023).
[28] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2023. Bloom: A 176b-parameter open-access multilingual language model. (2023).
[29] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110* (2022).
[30] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889* (2023).
[31] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems* 36 (2023), 21702–21720.
[32] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2023. Recent advances in natural language processing via large pre-trained language models: A survey. *Comput. Surveys* 56, 2 (2023), 1–40.
[33] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911* (2022).
[34] Shashi Narayan, Shay B Cohen, and Mirella Lapata. 2018. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *arXiv preprint arXiv:1808.08745* (2018).
[35] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.
[36] OpenAI(2023). 2024. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2024).
[37] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. 2024. AttAcc! Unleashing the Power of

PIM for Batched Transformer-based Generative Model Inference. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 103–119.
[38] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. *arXiv preprint arXiv:2311.18677* (2024).
[39] Jonathan Pilault, Raymond Li, Sandeep Subramanian, and Christopher Pal. 2020. On extractive and abstractive neural document summarization with transformer language models. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*. 9308–9319.
[40] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. 2020. Training Large Neural Networks with Constant Memory using a New Execution Algorithm. *arXiv preprint arXiv:2002.05645* (2020).
[41] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–14.
[42] Debendra Das Sharma. 2022. Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing With Heterogeneous Memory Hierarchy. *IEEE Micro* 43, 2 (2022), 99–109.
[43] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *Proceedings of the International Conference on Machine Learning (ICML)*. 31094–31116.
[44] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053* (2020).
[45] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2023. Powerinfer: Fast large language model serving with a consumer-grade gpu. *arXiv preprint arXiv:2312.12456* (2023).
[46] Trevor Strohman, Donald Metzler, Howard Turtle, and W Bruce Croft. 2005. Indri: A language model-based search engine for complex queries. In *Proceedings of the international conference on intelligent analysis*, Vol. 2. Washington, DC., 2–6.
[47] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. 2024. A Simple and Effective Pruning Approach for Large Language Models. *arXiv preprint arXiv:2306.11695* (2024).
[48] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 105–121.
[49] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Faerrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288* (2023).
[50] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*. PMLR, 38087–38099.
[51] Sungmin Yun, Kwanhee Kyung, Juhwan Cho, Jaewan Choi, Jongmin Kim, Byeongho Kim, Sukhan Lee, Kyomin Sohn, and Jung Ho Ahn. 2024. Duplex: A Device for Large Language Models with Mixture of Experts, Grouped Query Attention, and Continuous Batching. *arXiv preprint arXiv:2409.01141* (2024).
[52] Munazza Zaib, Quan Z Sheng, and Wei Emma Zhang. 2020. A short survey of pre-trained language models for conversational ai-a new age in nlp. In *Proceedings of the Australasian computer science week multiconference*. 1–4.
[53] 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. 2022. OPT: Open Pre-trained Transformer Language Models. *arXiv preprint arXiv:2205.01068* (2022).
[54] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *Symposium on Operating Systems Design and Implementation*.

# A Artifact Appendix

## A.1 Abstract

This artifact appendix provides a guideline for using LIA for LLM inference and how to reproduce two key results of this paper: 1) LIA's performance comparison to IPEX and FlexGen for online/offline inference and 2) LIA's CXL-offloading for large-batch inference. The following subsections outline the steps to access, build, and setup the software environment to deploy LIA on an AMX-supported CPU system with an NVIDIA GPU.

## A.2 Artifact Check-list (Meta Information)

- **Runtime environment:** Ubuntu 22.04.4 LTS, Linux kernel version 6.8.
- **Hardware:** Intel Xeon Platinum 8460H Processor, NVIDIA A100 GPU, and two Samsung CXL Type-3 memory expanders.
- **Required Disk Space:** 380 GB, 20 GB for the Docker image and the remaining for the model weights of OPT-30B and OPT-175B.
- **Output:** stdout.log from LIA, IPEX, and FlexGen programs, containing the inference latency.
- **Metrics:** End-to-end inference latency (s) and inference throughput (tokens/s) for online and offline inference, respectively.
- **Installation time:** 1 hour.
- **Experiment duration:** 13 hours.
- **Experiments reproduced:** SPR-A100 results of Figure 10 and Figure 11, and Table 3.
- **Publicly available:** Yes.
- **Code license:** Apache-2.0 license.
- **Archived:** https://zenodo.org/records/15104666

## A.3 Description

*A.3.1 How to Access.* The source code, scripts, and guidelines of LIA are publicly available on GitHub (https://github.com/Hyungyo1/LIA_AMXGPU) and Zenodo (https://zenodo.org/records/15104666). Other frameworks that we compare with is available at:

- **Intel Extension for PyTorch:** https://github.com/intel/intel-extension-for-pytorch.git
- **FlexGen:** https://github.com/FMInference/FlexLLMGen
- **PowerInfer:** https://github.com/SJTU-IPADS/PowerInfer

*A.3.1 Hardware Dependencies.* To run LIA for LLM inference, an AMX-supported CPU (later than 4$^{th}$ generation Xeon CPUs) system with an NVIDIA GPU and two or more CXL type-3 memory expanders are required. We use the following command to convert the CXL memory from device DAX mode to system RAM mode.

```
$ sudo daxctl reconfigure-device --mode=system-ram dax0.0
```

*A.3.1 Software Dependencies.* We provide a Dockerfile that modifies the one provided by Intel Extension for PyTorch. During the build of the Docker image, IPEX binds with `pytorch-cuda` to ensure NVIDIA GPU support in the environment.

## A.4 Installation

First, download the Github repository, initialize, and update the submodules as follows:

```
$ git clone https://github.com/Hyungyo1/LIA_AMXGPU.git
$ cd LIA_AMXGPU
$ git submodule sync
$ git submodule update --init --recursive
```

Alternatively, download the Docker image directly:

```
$ docker pull hyungyo/lia-amxgpu:latest
```

Next, build a docker image and compile LIA from source:

```
$ DOCKER_BUILDKIT=1 docker build -f \
    examples/cpu/inference/python/llm/Dockerfile \
    --build-arg COMPILE=ON -t lia-amxgpu:main .
```

Run the docker image with GPUs and mounted storage. Then, activate environment:

```
$ docker run --rm -it --gpus all --privileged -v \
    {storage_dir}:/home/storage lia-amxgpu:main bash
$ cd llm && source ./tools/env_activate.sh
```

## A.5 Experiment Flow

First, create dummy weights for OPT-175B model with the following command:

```
$ mkdir -p "/home/storage/opt-175b"
$ python utils/opt_dummy_weights.py --model-"opt-175b"
    --save_dir="/home/storage/opt-175b/"
$ cp /home/ubuntu/llm/utils/tokenizer/*
    /home/storage/opt-175b/
```

Then, run the following script to collect data for LIA and IPEX for online and offline inference:

```
$ bash scripts/run_performance.sh
```

For the experiments of LIA's CXL-offloading for offline inference, run the following script which will save the results to your mounted storage directory:

```
$ bash scripts/cxl_offloading.sh
```

The online and offline inference performance of FlexGen can be collected by running the following script in the Zenodo archived artifact:

```
$ cd FlexLLMGen
$ pip install torch
$ pip install -e .
$ cd flexllmgen && bash flexgen_ae_script.sh
```

## A.6 Evaluation and Expected Results

The key experiments in our paper will be reproduced: (1) the inference latency and throughput of LIA, IPEX, and FlexGen for online (Figure 10) and offline (Figure 11) inference, respectively and (2) LIA's performance with CXL-offloading (Table 3). After each experiment, the result data can be automatically collected to produce each figure and table with the following command:

```
$ python plots/generate_fig10_fig11.py
$ python plots/generate_tab3.py
```