2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA) | 979-8-3503-2658-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISCA59077.2024.00082

LLMCompass: Enabling Efficient Hardware Design for Large Language Model Inference

Hengrui Zhang August Ning Rohan Baskar Prabhakar David Wentzlaff
Princeton University
Princeton, New Jersey, USA
{hengrui.zhang, aning, rohanbp, wentzlaf}@princeton.edu

***Abstract*—The past year has witnessed the increasing popularity of Large Language Models (LLMs). Their unprecedented scale and associated high hardware cost have impeded their broader adoption, calling for efficient hardware designs. With the large hardware needed to simply run LLM inference, evaluating different hardware designs becomes a new bottleneck.**

This work introduces LLMCompass[1], a hardware evaluation framework for LLM inference workloads. LLMCompass is fast, accurate, versatile, and able to describe and evaluate different hardware designs. LLMCompass includes a mapper to automatically find performance-optimal mapping and scheduling. It also incorporates an area-based cost model to help architects reason about their design choices. Compared to real-world hardware, LLMCompass' estimated latency achieves an average 10.9% error rate across various operators with various input sizes and an average 4.1% error rate for LLM inference. With LLMCompass, simulating a 4-NVIDIA A100 GPU node running GPT-3 175B inference can be done within 16 minutes on commodity hardware, including 26,400 rounds of the mapper's parameter search.

With the aid of LLMCompass, this work draws architectural implications and explores new cost-effective hardware designs. By reducing the compute capability or replacing High Bandwidth Memory (HBM) with traditional DRAM, these new designs can achieve as much as 3.41x improvement in performance/cost compared to an NVIDIA A100, making them promising choices for democratizing LLMs.

***Index Terms*—Large language model, performance model, area model, cost model, accelerator**

I. Introduction

Large Language Models (LLMs), the technology behind OpenAI ChatGPT [49], Github Copilot [22], and Google Bard [24], are gaining widespread attention from the whole society. The capability of LLMs is related to their model size [29], [31], and larger models [8], [11] show impressive abilities [77] compared to smaller counterparts [16], [57], with future models expected to exceed trillions of parameters [17].

This unprecedented scale of LLMs poses challenges to deployment. Serving a GPT-3 (175B parameters) inference requires a minimum of five NVIDIA A100s solely to accommodate the model parameters (in half precision). This substantial hardware cost impedes the broader adoption of LLMs and motivates computer architects to design more cost-effective hardware. We identify three challenges that exist in designing hardware for LLM inference:

Lack of tools to evaluate hardware designs. Before diving into writing the RTL code, hardware designers may want to first sketch and compare different design choices. There are many properties we want for such a hardware evaluation tool before writing RTL. ① **Fast and accurate.** Due to the intense compute and memory hardware demand required for LLM inference, this tool needs to be as fast as possible without sacrificing accuracy. ② **Architecturally descriptive.** This tool should be general enough to describe different design choices: If it only applies to a specific architecture, the design space for computer architects will be limited. ③ **Performance-optimal.** The hardware performance is also affected by how the software is programmed (*e.g.*, how to map the workload to the hardware). The evaluation tool should optimize this software domain to fully demonstrate the hardware capability of each design. ④ **Cost-aware.** We also want to know how different hardware design choices affect the hardware cost to reason about cost-performance trade-offs.

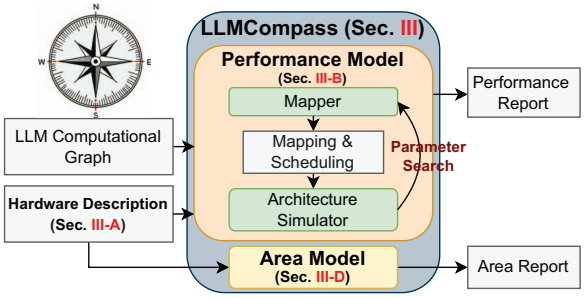


Fig. 1: An Overview of LLMCompass. LLMCompass can aid the hardware design process as a versatile evaluation tool.

Existing tools fail to meet these requirements. Roofline model analysis is fast but not accurate, and cycle-level simulators are accurate but slow. FPGA emulation is accurate and provides area statistics but requires significant engineering effort. To evaluate large-scale hardware designs in the era of LLMs, a new hardware evaluation tool is needed.

Lack of knowledge on how different hardware design choices affect LLM inference performance. As an emerging application, the hardware characteristics of LLMs remain to be understood. Besides the large volume of compute and memory requirements, LLMs are also unique in their auto-regressive way of generating tokens. We are interested in exploring whether these properties of LLMs will change common architecture wisdom.

[1] Available at https://github.com/PrincetonUniversity/LLMCompass.

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00082

Lack of cost-effective hardware designs to democratize LLMs. LLMs are powerful and capable, but are cost-prohibitive to deploy. To serve GPT-3, a DGX A100 compute node can cost over $100,000 USD [46], with each NVIDIA A100 featuring 54B transistors and 80 GB of High Bandwidth Memory (HBM). This high hardware cost hinders democratizing LLMs.

In this paper, we tackle these challenges and make three main contributions.

(1) We introduce LLMCompass, a hardware evaluation framework for LLM inference workloads (Sec. III). LLMCompass leverages the fact that mainstream ML hardware platforms share many architectural commonalities, allowing us to develop a general hardware description template for them. We also observe LLMs' computational graphs are composed of dense operators: matrix multiplication, softmax, layer normalization, *etc.*, all of which have a structural and hence predictable compute and memory access pattern. This allows LLMCompass to perform faster, higher-level tile-by-tile (block-by-block) simulations without losing accuracy compared to cycle-accurate simulators. The framework implements a mapper to manually manage the memory hierarchy and find the performance-optimal mapping and schedule scheme for dense workloads. LLMCompass also features a cost and area model based on public parameters to help designers reason about different design choices.

LLMCompass is validated on three commercial hardware designs: NVIDIA A100 [48], AMD MI210 [2], and Google TPUv3 [30], [45]. Compared to real-world hardware, LLMCompass' estimated latency achieves 10.9% error rate across various operators with various input sizes and 4.1% error rate for LLM inference. Implemented in Python, LLMCompass is still fast. It takes only 15-16 minutes to simulate a 4-A100 GPU node running GPT-3 175B inference, including 26,400 rounds of the mapper's parameter search (Figure 5i, tested on one core of Intel Xeon Gold 6242R CPU @ 3.10GHz).

(2) We leverage LLMCompass to draw architectural implications and explore how hardware design choices affect LLM inference (Sec. IV). We find that *prefill* and *decoding* pose different hardware requirements. *Prefill* can significantly benefit from more compute capability and buffers, while *decoding* barely gains from these and is more sensitive to memory bandwidth. These insights inspire us to think about new hardware design paradigms.

(3) We propose two cost-effective hardware designs different from conventional wisdom (Sec. V). We find that today's hardware design paradigms tend to fit massive compute capability and SRAMs in a huge die connected to high-end HBMs. We analyze the LLM inference characteristics and show how current hardware designs are inefficient. ① As LLM inference is mostly IO-bound, HBMs can be used to achieve low latency. However, HBM memory capacity limits the batch size, making it hard to fully utilize the massive compute capability. Based on this observation, we find that 95.3% of the original performance can still be achieved even if we prune the compute capability and buffer size by half. ② Larger batch size can significantly improve throughput as the model parameters are only read once for the whole batch. As memory capacity limits the batch size therefore limiting throughput, we propose to replace HBMs with traditional DRAM. We find that a larger batch size can compensate for the loss in memory bandwidth and can bring a 1.42x improvement in throughput and a 3.41x improvement in performance/cost.

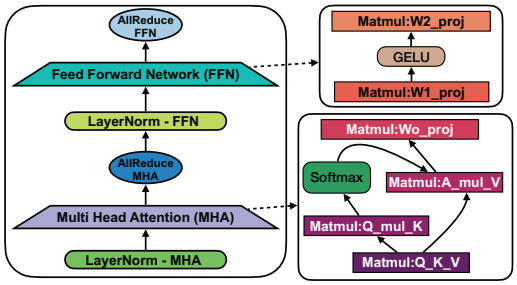


Fig. 2: A Decoder-Only Transformer Layer with Tensor Parallelism. GPT-3 175B [8] consists of a stack of 96 such layers.

II. Background

A. Large Language Models and Transformers

Large Language Models are variations of Transformer models [73] with a considerable amount of parameters that have been pre-trained on large corpora of data [40]. Today's LLMs can have as much as one trillion parameters [17]. Compared to smaller models, larger models (*e.g.* GPT-3 175B [8]) showcase a remarkable set of capabilities such as emergent abilities [77] and few-shot learning [8]. This increase in model size and the consequent memory and compute requirements have posed unique challenges for hardware.

We focus on Decoder-only Transformer models [55], which is the architecture adopted by most of the LLMs today: LLaMA [70], GPTs [8], [57], Bloom [80], PaLM [11], *etc.* The basic building blocks of these models are Transformer layers. As illustrated in Figure 2, each layer comprises a Multi-Head Attention block followed by an MLP block. These layers are then stacked together, forming the bulk of an LLM's memory and compute requirement. Transformers also use learned Vocabulary and Position embeddings, but for large models like GPT-3, these do not contribute significantly to either the memory or compute requirement ($< 2\%$). Without losing generality, we focus on Multi-Head Attention Transformers (GPT-style). There are other variations such as Multi-Query Attention [11], Mixture-of-Experts [17], and parallel Attention and MLP [11]. LLMCompass seamlessly supports all these possible variations as they share a common set of operators.

B. LLM Inference

Given an input prompt and the required number of output tokens, LLM inference can be divided into two stages [56]. ① *Prefill*: Processing the input prompt and computing the KV cache. The Key Value (KV) cache refers to the stored Key and Value tensors of the Attention block in each layer [56].

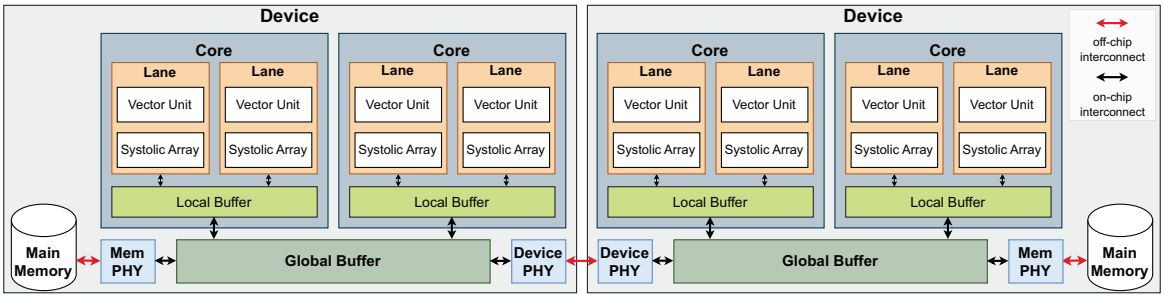


Fig. 3: LLMCompass' Hardware Description Template. In this example, each device has 2 cores and each core has 2 lanes.

② *Decoding*: Generating output tokens one after another in an auto-regressive manner: The Key and Value of the newly generated token will be concatenated to the KV cache and used for generating the next token. The latency of *prefill* and *decoding* is mostly determined by the input and output sequence lengths, respectively. In *prefill*, as the entire input sequence needs to be multiplied by all the parameters, it is usually bounded by compute. In *decoding*, each new token needs to be multiplied by all the parameters and concatenated to the KV cache, so *decoding* is usually bounded by reading parameters and KV cache.

Latency and throughput are the key metrics to evaluate LLM inference systems. For interactive use cases such as chatbots [49], it is imperative to optimize latency. For background data processing use cases such as data wrangling [42] or form processing [9], throughput is more important. The tradeoff between latency and throughput is determined by batch size: larger batch increases throughput at the cost of higher latency.

C. Parallelizing LLM Inference

Due to the large volume of compute and memory operations, it is beneficial to parallelize LLM inference across multiple devices. This leads to much better performance and can be necessary if the model's parameters along with the KV cache do not fit in a single device's memory. For LLM inference, there are two model parallelization schemes: pipeline parallelism and tensor parallelism. In pipeline parallelism, different layers of the model are grouped into sequential partitions and assigned to different devices like a hardware pipeline. This scheme has the effect of considerably increasing throughput at the expense of increased latency. On the other hand, tensor parallelism, as proposed by Megatron-LM [64], partitions each layer of the model across the available devices, thereby decreasing latency at the cost of frequent device-device communication and synchronization. As shown in Figure 2, this scheme requires two *all-reduce* for each Transformer layer, one after the Attention block and another after the MLP block.

III. LLMCompass

An overview of LLMCompass (**L**arge **L**anguage **M**odel **Com**putation **P**erformance and **A**rea **S**ynthesi**s**) is shown in Figure 1. To evaluate the performance (*e.g.*, throughput and latency) of running a Transformer-based large language model on a hardware system, two inputs are needed: the computational graph of the LLM and a **hardware description** (Section III-A). Given the input, the **performance model** (Section III-B) generates a performance report. The **mapper** conducts a parameter search along with the **architecture simulator** to find the best mapping and scheduling scheme. At the same time, the **area model** (Section III-D) generates the area and cost report.

TABLE I: Examples of LLMCompass's Hardware Description

Key Specifications	NVIDIA A100 [48]	AMD MI210 [2]	Google TPUv3[2] [45]
Frequency (MHz)	1410	1700	940
Core count	108	104	2
Lane count	4	4	1
Vector width	32	16	4×128
Systolic array	16×16	16×16	128×128
Local buffer (KB)	192	80	8192
Global buffer (MB)	40	8	16384
Global buffer (bytes/clk)	5120	4096	490
Memory bandwidth (TB/s)	2	1.6	-
Memory capacity (GB)	80	64	-
Device-device bandwidth (GB/s)	600	300	162.5

A. Hardware Description Template

The hardware description template of LLMCompass is introduced below, as shown in Figure 3:

- A **system** (*e.g.*, a DGX node) is composed of multiple devices connected through a device-device interconnect (*e.g.*, NVLink or Infinity Link).
- Each **device** (*e.g.*, a GPU) is composed of multiple cores, a shared global buffer, and an off-chip main memory. The **global buffer** (*e.g.*, L2 cache in NVIDIA GPUs) is connected to the main memory, device-device interconnect, and all the cores.
- Each **core** (*e.g.*, a Stream Multiprocessor in NVIDIA GPUs) can have multiple lanes sharing a **local buffer** (*e.g.*, L1 cache in NVIDIA GPUs). The local buffer is connected to the global buffer through the on-chip interconnect.
- Each **lane** is independent from each other and has its own **vector unit**, **systolic array**, registers and control logic.

[2]One TPUv3 core. Each TPUv3 chip has two TPUv3 cores. TPUv3 cores within the same chip are connected by internal links.

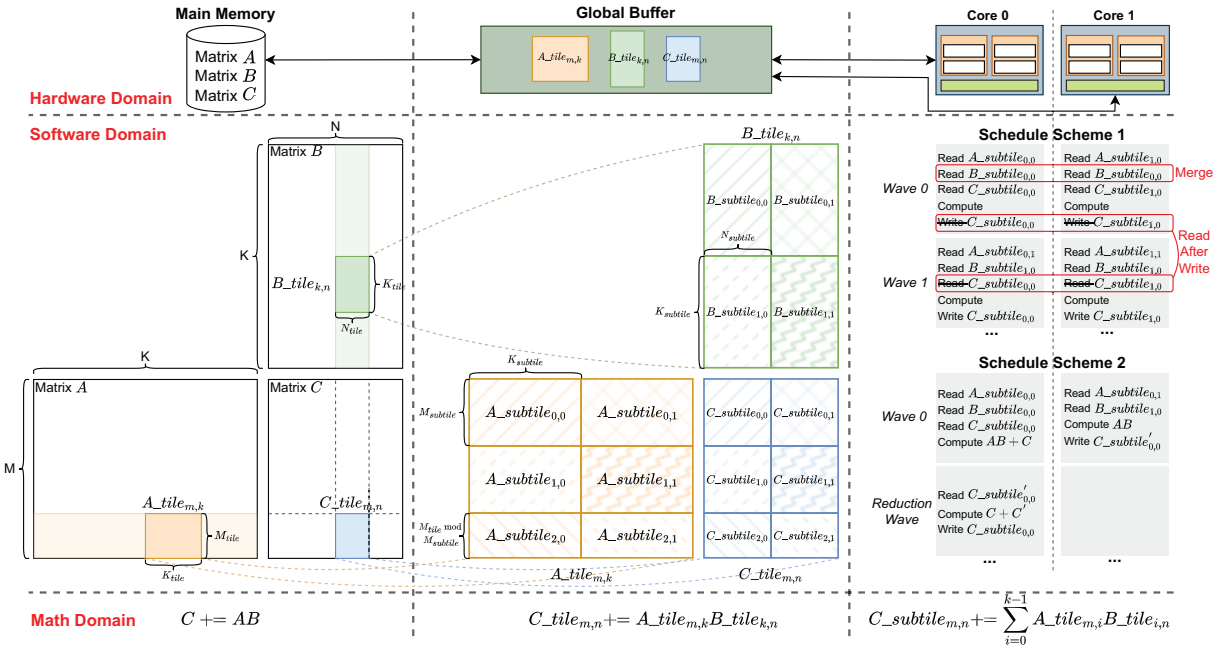


Fig. 4: Visualization of a Matrix Multiplication in LLMCompass as in Section III-B1.

In existing devices, the local and global buffers are usually on-chip SRAM: cache, scratchpad, or a combination of both. LLMCompass doesn't distinguish between cache and scratchpad because the memory is explicitly managed by the mapper. We believe this assumption does not lose generality as a highly optimized library will also carefully manage the memory. The main memory is usually off-chip DRAM: HBM, DDR memory, CXL memory, *etc*, all of which can be described by our parameterized hardware description template.

We find this hardware description is general enough to describe the mainstream machine learning platforms of today: NVIDIA GPUs, AMD GPUs, and Google TPUs, as shown in Table I with a sample of key specifications listed. It is also flexible enough to explore future architectures.

B. Performance Model

The computational graph of a Transformer is composed of a stack of Transformer layers. Each layer is composed of a series of operators, including matrix multiplication (*Matmul*), *Softmax*, layer normalization (*LayerNorm*), and activation functions (*e.g.*, GELU [28] as in GPTs [8], [57]). In a multi-device setup, communication primitives such as *all-reduce* operators are also needed to perform tensor parallelism. The key challenge is how to simulate the performance of different operators and communication primitives on a given hardware system - this requires knowledge about the hardware and how to map and schedule operators on a multi-level compute system with a multi-level memory hierarchy.

To solve this, LLMCompass introduces a mapper and an architecture simulator to build a performance model. Conceptually, we simulate running an operator on the chosen hardware in a recursive manner: we first partition the problem into smaller sub-problems that can fit in the global buffer. The sub-problem is then divided into smaller sub-sub-problems that can fit in each core's local buffer. The partitioning, mapping, and scheduling are generated by the mapper and a parameter search is conducted to find the optimal mapping and scheduling. LLMCompass always tries to find the performance-optimal mapping to fully demonstrate the hardware capability.

*1) **Matrix Multiplication**:* The process of simulating a matrix multiplication is visualized in Figure 4. $\mathbf{A}$ is a $M \times K$ matrix with M rows and K columns. Similarly, $\mathbf{B}$ and $\mathbf{C}$ are $K \times N$ and $M \times N$ matrices respectively. A generalized matrix multiplication is defined as $\mathbf{C} = \mathbf{AB} + \mathbf{C}$.

From main memory to global buffer: To maximize data reuse, matrix multiplication is usually calculated in a tile-by-tile manner [34]. As shown on the left of Figure 4, matrix $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are divided into tiles small enough to fit into the global buffer. In each step, one $A\_tile_{m,k}$, $B\_tile_{k,n}$, and $C\_tile_{m,n}$ are read into the global buffer, the cores then perform the computation, and the results are written back.

From global buffer to local buffer: With tiles inside the global buffer, we now need to parallelize the computation of $C\_tile_{m,n} = A\_tile_{m,k} B\_tile_{k,n} + C\_tile_{m,n}$ on multiple cores. As shown in the middle of Figure 4, these tiles are further divided into smaller sub-tiles to fit in each core's local buffer. It then becomes a scheduling problem to map sub-tiles onto cores.

The right of Figure 4 shows two possible schedule schemes:

- **Schedule Scheme 1:** Different cores working on different $C_subtile$s in the same column. At *wave 0*, as *core 0* and *core 1* both need to read the same $B\_subtile$, their memory access to the global buffer should be merged. In our simulator, this memory access merging is automatically identified and taken care of. As the same core keeps updating the same $C\_subtile$, there is no need to first write the partial result and then read it from the global buffer. This *Read-After-Write* dependency is also automatically taken care of by the simulator.
- **Schedule Scheme 2:** Different cores working on the same $C\_subtile$. *Core 0* and *core 1* first read the data and calculate the partial results, then perform a reduction and write back the final results.

In reality, with more cores and more tiles, the schedule space can be more complicated than the example shown in Figure 4.

From local buffer to lanes: Similarly, within each core, the sub-tiles are further partitioned into sub-sub-tiles to be mapped to lanes sharing a local buffer. After that, the sub-sub-tiles are finally passed to the systolic arrays. LLMCompass leverages SCALE-Sim [61], [62], a cycle-level systolic array simulator, to mimic the behavior of a systolic array and get the cycle count. LLMCompass caches the results of SCALE-Sim into a look-up table to avoid duplicated simulation. A reduction will be performed by the vector unit if needed.

Mapper: A parameter search is performed by the mapper to determine the best tiling scheme and schedule scheme. To overlap computation with memory accesses, we also add software pipelines (double buffering) at each level of the memory hierarchy as scheduling options. The downside of enabling software pipeline is that it requires extra buffer space so the maximal tile size will be reduced, causing potentially lower utilization of systolic arrays. However, we find software pipeline to be beneficial in most cases.

*2) **Communication Primitives**:* We use the link model as in AHEAD [1] and LogGP [4]. Suppose L is the link latency, O is the additional overhead associated with the data transfer, and B is the link bandwidth. The latency T to transfer n bytes of data through a link is expressed in Equation 1 and 2:

$$T = L + O + \frac{\hat{n}}{B} \tag{1}$$

$$\hat{n} = \left\lceil \frac{n}{MaxPayload} \right\rceil * Flit\_size + n \tag{2}$$

On top of this, we implement ring all-reduce [52], which is a bandwidth-optimal all-reduce algorithm. We use a 16-byte $Flit\_size$ and a 256-byte $MaxPayload$ based on NVLinks [18]. We don't model more communication primitives as LLM inference only requires *all-reduce* for tensor parallelism and *peer-to-peer* for pipeline parallelism.

*3) **Other Operators**:* We also model *Softmax*, *LayerNorm*, and *GELU* following a similar methodology as in SectionIII-B1. The differences are as follows: ① These operators have fewer dimensions and are therefore simpler: *Softmax* and *LayerNorm* operate on two-dimensional data, and *GELU* operates on one-dimensional data, while *Matmul* operates on three-dimensional data. As each dimension requires tiling and scheduling, the mapper search space is much smaller. ② They do not use systolic arrays. ③ *Softmax* and *LayerNorm* involves reductions to calculate the sum, mean, or max. Therefore, the schedule scheme needs to consider that the reduction can be either performed within one core or might be splitted across different cores. For the reduction within each core, a reduction tree is implemented. Inter-core reduction is implemented with atomic operations. *Softmax* is implemented with the online algorithm [39]. *GELU* is approximated with $tanh$ [28].

C. Performance Model Validation

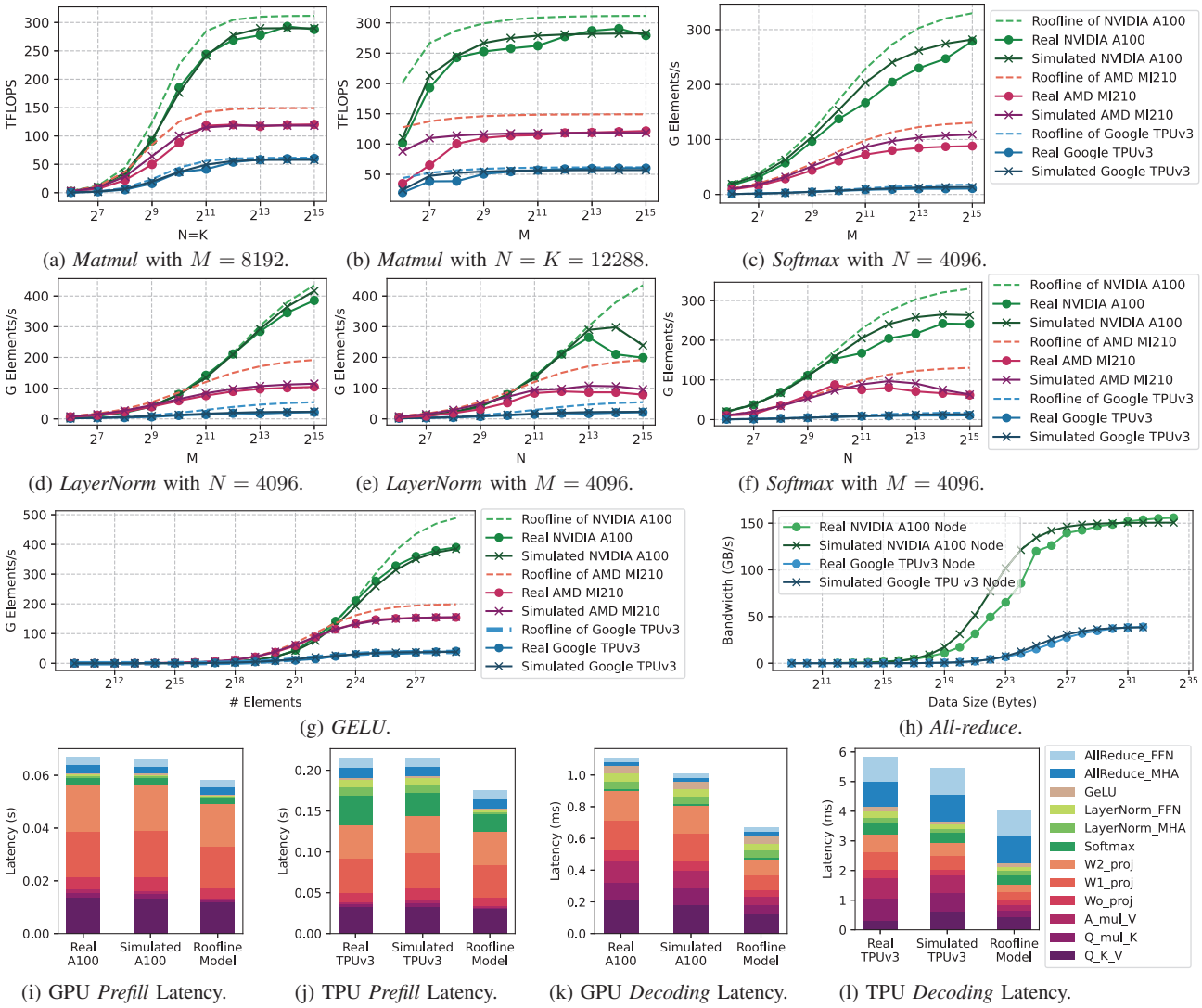
In this section, we validate our framework against three real hardware platforms: (1) a datacenter GPU node with 4 NVIDIA A100 SXM4 GPUs (80 GB) fully connected by NVLinks; (2) a Google Cloud TPU node with 8 TPUv3 cores connected in a 2D torus topology; (3) an AMD MI210 GPU[3]. The results are shown in Figure 5. For NVIDIA GPUs, CUDA 11.7 and PyTorch 2.0 are used to benchmark operators in half precision (FP16) with `torch.compile` enabled for *LayerNorm* and *GELU* to maximize performance. Communication primitive *all-reduce* is benchmarked with nccl-tests [43], a communication primitive performance benchmark for NVIDIA GPUs. For Google TPUs, JAX 0.4.18 is used to benchmark operators and communication primitives. Due to the hardware feature of TPUs, *Matmul* is benchmarked in bfloat16 (BF16) and all the other operators are in FP32. For AMD GPU, ROCm 5.4.2 and PyTorch 2.0 are used along with FP16 for *Matmul* and FP32 for other operators. The kernel launch overhead including the framework overhead is measured by running the operator with an input of size 1.

As shown in Figure 5, for *Matmul*, *Softmax*, *LayerNorm*, *GELU*, and *all-reduce*, LLMCompass achieves an average error rate of 9.0%, 12.0%, 13.8%, 5.0%, and 14.9% respectively. For LLM inference, LLMCompass achieves an average error rate of 0.69% and 7.5% for *prefill* and *decoding* respectively. **On average, LLMCompass achieves a 10.9% error rate for different operators at various input sizes and a 4.1% error rate across the *prefill* and *decoding* stages.**

GELU is more accurate than other operators because it is element-wise and easy to simulate. *Layernorm* and *Softmax* are less accurate because of the reduction involved. *All-reduce* is less accurate probably because of unidealed hardware. Matrix multiplication is accurate (except for small ones on AMD MI210 as in Figure 5b) because it is highly optimized on those hardware platforms. As matrix multiplication is the dominant part of most of the models today, a validity of performance across different types of models can be achieved.

Although not a perfect match to real-world hardware, LLMCompass is able to show a similar trend that a naive roofline model fails to show. For example, in Figure 5e, as the reduction dimension of *LayerNorm* increases to an extreme, the throughput should drop due to the increasing reduction cost. LLMCompass is able to catch this trend.

[3]We set the frequency to 1400 Mhz to avoid frequency fluctuation



(a) *Matmul* with $M = 8192$. (b) *Matmul* with $N = K = 12288$. (c) *Softmax* with $N = 4096$.

(d) *LayerNorm* with $N = 4096$. (e) *LayerNorm* with $M = 4096$. (f) *Softmax* with $M = 4096$.

(g) *GELU*. (h) *All-reduce*.

(i) GPU *Prefill* Latency. (j) TPU *Prefill* Latency. (k) GPU *Decoding* Latency. (l) TPU *Decoding* Latency.

Fig. 5: Performance Model Validations. *Matmul* takes a $M \times K$ (M rows and K columns) and a $K \times N$ matrix as input. 12288 is the model dimension of GPT-3 [8]. *Softmax* and *LayerNorm* take a $M \times N$ matrix and perform normalization on the N dimension. *Prefill* latency is measured by running one layer of GPT-3 with batch size 8 and sequence length 2048. *Decoding* latency is per GPT-3 layer per output token and is measured by the latency of generating the 1024th output token with batch size 8 and input sequence length 2048. For (a)-(g), a single GPU/TPU device is used. For (h)-(l), the 4-A100 GPU node and 8-TPUv3-Core TPU node are used with tensor parallelism.

LLMCompass' results are totally interpretable without incorporating any fudge factor and we believe this interpretability is more important than perfectly matched results. Here are some possible causes of the mismatch between LLMCompass and real hardware:

- Lack of hardware knowledge. We have little knowledge about the micro-architecture of GPUs and TPUs (*e.g.*, hardware pipeline design or scheduler design). With a large input size, the hardware is well utilized and some overhead can be hidden. However, with a small input size, it's hard to hide the overhead and micro-architecture details affect performance significantly. Also, the Tensor Cores in NVIDIA GPUs and Matrix Cores in AMD GPUs are simulated as systolic arrays in LLMCompass, which may not be true in reality.
- Lack of software knowledge. We don't know how operators and communication primitives are implemented on these platforms as they are closed-source libraries. We conduct a thorough parameter search for each input size to maximize performance, but in reality those libraries probably use heuristics to determine mapping and scheduling, which may not be optimal at all input sizes (*e.g.*, we find that for a *Matmul* with $M = 64$ and $N = K = 12288$, AMD MI210 is less than 25%

of its roofline performance while a NVIDIA A100 can achieve 50% of its roofline performance.). Also, some key information is not available. For example, we cannot find the packet format for TPU-TPU communication and have to use the NVLink packet format instead.

- Non-ideal hardware. LLMCompass assumes a fixed frequency, but when testing real-world hardware, we have no control over the frequency of the datacenter GPU or TPU nodes. LLMCompass also assumes bandwidth can be utilized at full rate, but in reality there may be some other overhead (*e.g.*, error correction code).

D. Area and Cost Model

As chip designers increase die area to improve single chip performance, fewer chips fit per wafer and may also risk decreased yield, leading to increased costs. LLMCompass incorporates area and cost models to allow designers to reason about these performance-area trade-offs. These models use the provided hardware description with estimated transistor counts and/or die areas from known components to find the total device die area - our methodology is explained as follows.

Within each core's lanes, we estimate the vector units' and systolic arrays' transistor counts from open-source designs, tape-outs, and generators [20], [38], [83]. We estimate each lane's register file's area overhead using an empirical area model [58]. For the local buffer shared amongst lanes in each core as well as the global buffer shared amongst cores, we model them as SRAM caches and derive their areas using CACTI [41] and scale results down to a 7nm process. For memory and device-device interconnect, we estimate PHY and controller area based on annotated A100 and MI210 die photos [53], [65]. In our calculations, the controller area scales based on the process node, but the PHY area remains fixed as they do not scale well due to internal analog devices.

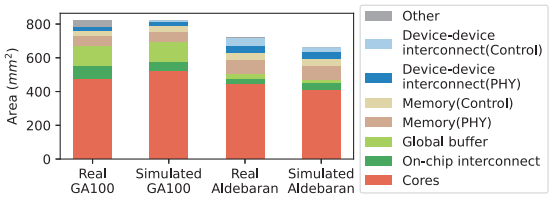
We account for extra per lane overheads (*e.g.*, control signals) by calculating the core area using our model and taking the difference from the expected die areas taken from annotated photos. We then divide the overhead per lane, per scheduler width (32 in A100s, 16 in MI210). Similarly, we account for extra per core overheads (*e.g.*, core-to-core crossbars) by calculating the expected die area with our model and splitting the area between the cores. These per-lane and per-core overhead estimates are averaged between AMD and NVIDIA chips.

To estimate cost, LLMCompass uses supply chain modeling [44] for wafer costs to calculate per-die costs. These per-die costs do not incorporate any IP, masks, or packaging costs. For memory costs, we use average DRAM spot prices for DDR [71] and consumer estimates for HBM2e [35].

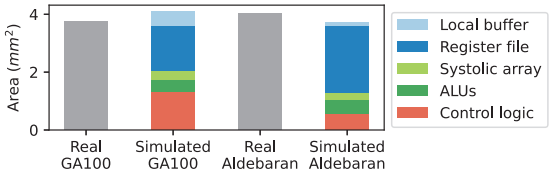
Table II shows a sample of the transistor counts and corresponding 7nm die areas of the parameters used in the area model. Using their respective architecture white papers, we model GA100 [48] (the die used in NVIDIA A100) and Aldebaran [2] (the die used in AMD MI210) dies to estimate their total die areas, shown in Fig. 6a. For the accounted-for components, LLMCompass' area model estimates for

TABLE II: A Sample of Area Model Parameters (7nm)

Parameter	Transistor Count	7nm Area (μm^2)
64 Bit Floating Point Unit	685300	7116
32 Bit Int ALU	177000	1838
Per Lane Overhead	996200	10344
Per Core Overhead	44300000	460000
1024 Bit HBM2e Control	552743000	5740000
1024 Bit HBM2e PHY	-	10450000



(a) Die Area Breakdown of NVIDIA GA100 and AMD Aldebaran.



(b) Core Area Breakdown (Stream Multiprocessor for NVIDIA GPUs and Compute Unit for AMD GPUs).

Fig. 6: Area Model Validations.

GA100 and Aldebaran dies achieve a 5.1% and 8.1% error respectively. We attribute these differences to the core's microarchitecture and core-to-core communication overheads which are proprietary and difficult to estimate. Our model also allows users to break down a single core's area into its individual components, shown in Fig. 6b.

IV. Architectural Implications

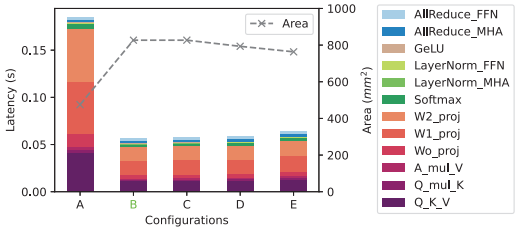
With LLMCompass, we are able to conduct a design space exploration and shed light on how to design efficient hardware systems for LLM inference. In this section, we use LLMCompass to study how different compute system configurations, memory bandwidth, and buffer sizes affect LLM inference performance and draw architectural implications. These insights inspire us to propose new designs as in Section V.

A. Experimental Setup

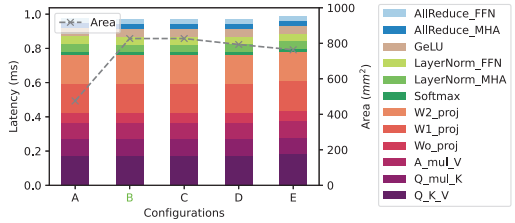
For all the unmentioned specifications, we use the specifications of an NVIDIA A100 (as in Table I) and 4-way tensor parallelism. *Prefill* latency (also know as TTFT, time to first token) is measured by running one GPT-3 layer with batch size 8 (a balancing point between latency and throughput) and input sequence length 2048 (a medium-long sequence for GPT-3). *Decoding* latency (also know as TBT, time between tokens) is measured as the latency of generating the 1024th output token when running one GPT-3 layer with batch size 8 and input sequence length 2048. We use FP16 for all the operators.

TABLE III: Five Compute System Designs.

Specifications	A	B	C	D	E
Core count	128	128	128	32	8
Lane count	4	4	1	1	1
Vector width	8	32	128	512	2048
Systolic array	8×8	16×16	32×32	64×64	128×128
Local buffer (KB)	192	192	192	768	3072



(a) *Prefill* Latency (TTFT) per GPT-3 Layer.



(b) *Decoding* Latency (TBT) per GPT-3 Layer per Output Token.

Fig. 7: Impact of Compute System Design on Performance.

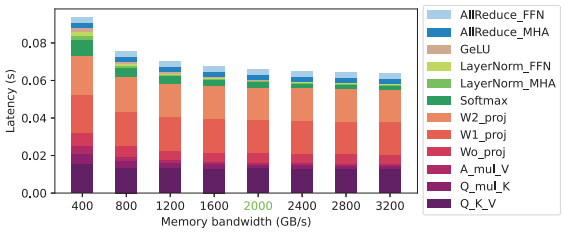
B. Compute System

We test five different compute system designs as shown in Table III. From A to E, we increase each core's systolic array, vector unit, and local buffer capacities. B represents a full GA100. We keep B, C, D, and E to have the same total compute capability and total buffer size to compare the design choice of fewer big cores or more tiny cores. Configuration A only has a quarter of the compute capability compared to others. All the designs have the same amount of total buffer size and register file size scales with vector width.

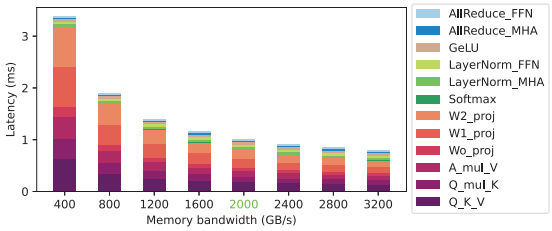
Figure 7 shows *prefill* and *decoding* latencies for these designs. Compared to the GA100, design A has 3.25x higher *prefill* latency but is only 0.1% slower at *decoding* and uses only 57.8% of the area. Design E with the largest cores see *prefill* and *decoding* latency increase by 12.4% and 1.9% respectively, but can reduce die area up to 7.7%.

Analysis: For the *prefill* stage, B is much faster than A because *prefill* is compute-bound. As per core systolic arrays and vector units scale, the tile size needs to increase to fully utilize larger computing units. Bigger tiles can cause more padding as the problem size needs to be quantized to the tile size and hardware size. Although large systolic arrays and vector units can be more area-efficient, they are harder to schedule and fully utilize.

Since *decoding* is IO-bound, increasing compute capability barely helps, which explains why A and B have similar performance. As the matrix multiplications during *decoding* are narrow (*e.g.* 16×12288), it is even harder to fully utilize larger systolic arrays/vector units and performance degrades.



(a) *Prefill* Latency (TTFT) per GPT-3 Layer.



(b) *Decoding* Latency (TBT) per GPT-3 Layer per Output Token.

Fig. 8: Impact of Memory Bandwidth on Performance.

Implications:

① *Increasing compute capability significantly helps prefill but barely helps decoding.*

② *Larger systolic arrays and vector units are more area-efficient but harder to fully utilize.*

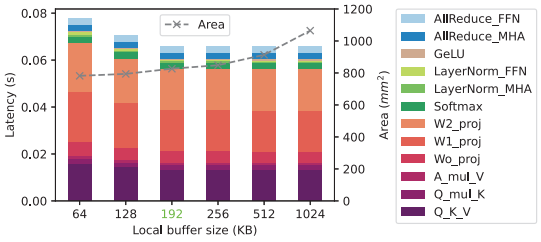
C. Main Memory

As main memory capacity is considered more of a constraint (enough capacity is required to hold the parameters and KV cache), we will focus on the impact of main memory bandwidth. Figure 8 details the performance results for sweeping memory bandwidth from 400 to 3200 GB/s. For *prefill*, increasing memory bandwidth from 800GB/s to 2000GB/s reduces latency by 14.3%, and further increasing to 3200GB/s has a marginal performance gain of 3.5%. For *decoding*, increasing from 800GB/s to 2000GB/s has a speedup of 1.88x, and further increasing to 3200GB/s brings another 26% gain.

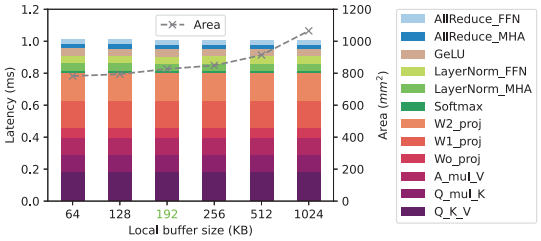
Analysis: In the *prefill* stage, *Matmul*s are significantly faster when increasing memory bandwidth from 400GB/s to 800GB/s. Further increasing bandwidth does not significantly affect *Matmul* performance as it becomes compute-bound. For IO-bound *GELU*, *LayerNorm*, and *Softmax*, larger memory bandwidth realizes significant speedup.

In the *decoding* stage, *Matmul*s are significantly faster with increased memory bandwidth, mainly because they are narrow (turn into a vector-matrix multiplication at batch size 1) and IO-bound. In this stage, *GELU*, *LayerNorm*, and *Softmax* have a small input size. They are dominated by kernel launch overhead and barely affected by memory bandwidth.

③ *Decoding is much more sensitive to memory bandwidth than prefill.*



(a) *Prefill* Latency (TTFT) per GPT-3 Layer.



(b) *Decoding* Latency (TBT) per GPT-3 Layer per Output Token.

Fig. 9: Impact of Local Buffer Size on Performance.

D. Local and Global Buffer

Local Buffer. We fix the hardware specifications to an NVIDIA A100 (as in Table I) and sweep local buffer size. The results are shown in Figure 9. For *prefill*, increasing the local buffer size from 64KB to 192KB improves the performance by 18.0% while increasing the area by 5.8%. Further increasing to 1024KB has a negligible performance gain of only 0.2% at the cost of 28.8% bigger area. For the *decoding* stage, increasing the local buffer size from 64KB to 1024KB only increases the performance by 0.5%.

Analysis: The reduced *prefill* latency with larger local buffers is mainly because of reduced matrix multiplication latencies. A larger local buffer enables larger matrix tiles and therefore higher systolic array utilization rate. A local buffer size of 192KB is just enough for matrix multiplication of $128\times128\times128$ at FP16 with double buffering technique. It can fully utilize the 16×16 systolic arrays, shedding some insight on the NVIDIA A100's design choices. Increasing local buffer size when the systolic array is already fully utilized leads to marginal performance gains. For *decoding* stage, increasing local buffer size does not help because it's IO-bound.

Global Buffer. The performance trends for global buffer size are similar to Figure 9. Increasing the global buffer size from 10MB to 40MB speeds up *prefill* by 11.8% while increasing area by 9.6%. Further increasing to 80MB only brings a performance gain of 0.01% at the cost of 11.7% bigger area. For *decoding*, increasing global buffer size from 10MB to 80MB has a performance gain of only 0.7%.

Analysis: Larger global buffers enable larger matrix tiles, increasing systolic array utilization and data reuse at the global buffer level. Similarly, increasing global buffer size has diminishing returns once the systolic arrays are saturated. The *decoding* stage is not bounded by computation so it barely benefits from the larger global buffer.

TABLE IV: Comparison with NVIDIA GA100

Specifications	Latency Design	GA100 (Full)	Throughput Design
Core count	**64**	128	**64**
Lane count	4	4	4
Vector width	32	32	32
Systolic array	16×16	16×16	$\mathbf{32\times32}$
Local buffer (KB)	192	192	**768**
Global buffer (MB)	**24**	48	48
Global buffer (bytes/clk)	**2560**	5120	5120
Memory bandwidth (TB/s)	2	2	**1**
Memory capacity (GB)	80	80	**512**
Memory protocol	HBM2E	HBM2E	**PCIE 5.0/CXL**
Die area (TSMC 7nm, mm^2)	478	826	787
Normalized performance	0.95	1	1.41
Estimated die cost	\$80	\$151	\$142
Estimated memory cost	\$560	\$560	\$154
Estimated total cost	\$640	\$711	\$296
Normalized performance/cost	**1.06**	1	**3.41**

④ *Large buffers help prefill but not decoding.*
⑤ *Buffers should be large enough to fully utilize the systolic arrays.*

V. Efficient Hardware Design with LLMCompass

Ideally, efficient hardware design will optimize for both performance and cost. This section draws from the insights in Section IV and proposes two efficient hardware designs: a latency-oriented design and a throughput-oriented design. Both of these designs aim to reduce hardware costs while maintaining or improving performance. The key specifications are shown in Table IV. All the other specifications (*e.g.*, frequency, register file size, device-device interconnect, kernel launch overhead, and framework overhead *etc.*) are the same as an NVIDIA GA100 for fair comparison.

A. Latency-Oriented Design

LLM inference latency means the total time between receiving the request and generating the last token. It is a critical metric for interactive use cases like chatbots. It is composed of *prefill* latency, the time to process the input sequence, and *decoding* latency, the time to generate the output sequence in an auto-regressive way. Inference latency is usually dominated by *decoding* unless the input sequence is much longer than the output sequence. *Decoding* is IO-intensive and is mostly bounded by reading model parameters and KV cache.

Observation: As latency is mostly IO-bound, memory bandwidth is the key to reducing latency, making HBM the best choice. However, due to the capacity limit of HBM, the batch size cannot be too large: the size of the KV cache and intermediate values is proportional to batch size. Therefore, the massive compute capability is not fully utilized.

Proposal: We propose an efficient latency-oriented design by pruning half of the compute capability while using the same memory system as a GA100, as shown in the left of Table IV.

Results: Compared to an NVIDIA GA100, the die area is reduced by 42.1% while keeping 95.3% of the performance on average. The results are shown in Figure 10.

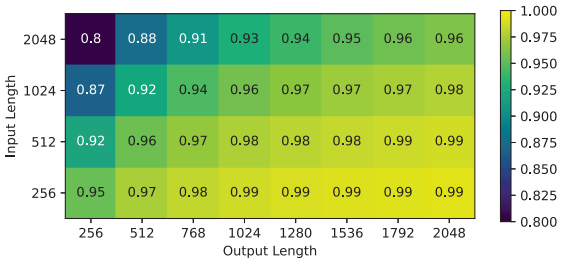


Fig. 10: End-to-End Performance of Latency-Oriented Design Normalized to GA100. Performance metric: inverse of latency (higher is better). Settings: batch size[4]16, 4-way tensor parallelism, running 48 GPT-3 layers (half of GPT-3).

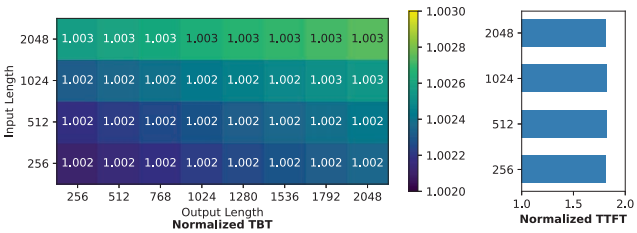
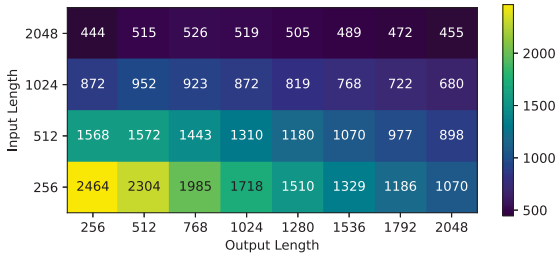
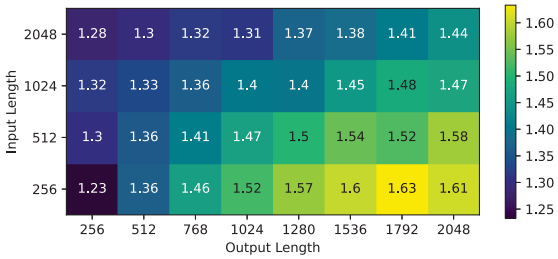


Fig. 11: TTFT and TBT of Latency-Oriented Design Normalized to GA100. Reducing compute capability by half barely hurts TBT but will bring 1.82x slowdown for TTFT.



(a) Throughput of Throughput-Oriented Design (Tokens/s).



(b) Normalized to a 8-GA100 GPU Node.

Fig. 12: End-to-End Performance of Throughput-Oriented Design. Performance metric: throughput. Settings: largest batch size within memory capacity, 8-way pipeline parallelism where each device runs 12 GPT-3 layers (1/8 of GPT-3).

Discussion: Due to the IO-bound *decoding* stage, the over-provisioned GA100 is not able to realize significantly improved inference performance compared to our latency-oriented design. As shown in Figure 11, our pruned design achieves identical *decoding* performance as a GA100. The GA100 is an enormous die and is susceptible to yield issues - A100 dies are already binned to have 108 functioning SMs out of 128. Our latency-oriented design shows that even with half the cores and SRAM disabled, the device can still achieve similar performance. This may motivate designers to salvage previously deemed faulty chips and manufacture them into separate products focused on LLM inference.

Pruning the compute capability only hurts the compute-bound *prefill* performance. As *prefill* is more dominant at long input sequence and short output sequence, the performance degradation will be more visible under these cases, which explains why we only achieve 80% of the GA100 performance at input length 2048 and output length 256. With a smaller input length and larger output length, our pruned latency-aware design can achieve 99% the performance as GA100.

B. Throughput-Oriented Design

For background use cases such as form processing or data wrangling, throughput can be more important than latency. There are generally two ways to improve throughput:

- Decrease latency - As latency is mostly IO-bound by reading parameters and KV cache, the best way to improve latency is to further improve memory bandwidth. As HBM is already expensive, this may not be easily achieved without increasing cost.
- Increase batch size - Generally, larger batch sizes are more efficient for throughput because the parameters are only read once for the whole batch. Larger batch sizes can also improve the hardware utilization rate. The downside is that a larger batch size consumes more compute power and increases KV cache accesses.

Observation: Increasing batch size is a more efficient way to improve throughput compared to decrease latency, which requires expensive high-end HBMs or even SRAMs. With a larger batch size, more memory capacity is needed to hold the larger KV cache and intermediate values.

Proposal: We propose a throughput-oriented design as shown in the right of Table IV. To hold larger batches, we use 512GB of DRAM powered by 256 PCIe 5.0 channels with an aggregated memory bandwidth of 1TB/s. (According to our area model, an $800mm^2$ die's perimeter is able to fit around 400 PCIe 5.0 channels.) Considering the high cost and limited capacity of HBMs, this design is more cost-effective. With larger batch sizes comes a greater need for compute capability, so we quadruple the systolic arrays and the local buffer. We halve the core count and vector unit to maintain a similar die area as GA100.

Results: Compared to an NVIDIA GA100, the die area is slightly smaller and the throughput is improved by 1.42x on average. The results are shown in Figure 12. By replacing HBMs with traditional DRAMs, the cost is reduced by 58.3%, making a total of 3.41x gain in performance/cost.

[4]In reality, a batch size of 16 with input length 2048 and output length 2048 will slightly exceed the memory capacity.

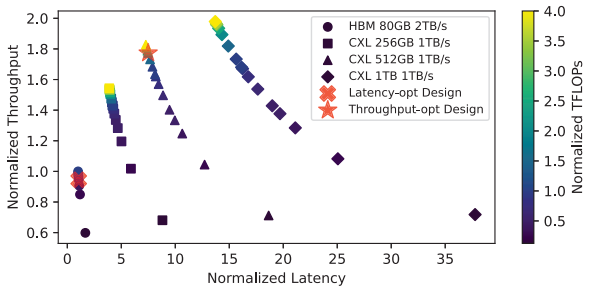


Fig. 13: Design Space Exploration with LLMCompass. The proposed latency-optimized design and throughput-optimized design are marked in red. Throughput and latency are normalized to a 4-GA100 node. Sweep parameters: compute system design, buffer size, memory type and capacity. Settings: largest batch size within the memory capacity, input length 1024, output length 1024, 4-way tensor parallelism, running 48 GPT-3 layers (half of GPT-3). It took 84 minutes to collect all the data points on one Intel Xeon Gold 6242R CPU @ 3.10GHz.

Discussion: Our design has 6.4x the memory capacity of a GA100, which allows more than 12x bigger batch size after subtracting the fixed space occupied by model parameters. Ideally, with half the bandwidth of a GA100, this configuration can achieve more than 6x improvement in throughput. However, batching only reduces model parameter accesses but not KV cache reads. With a much larger batch, KV cache accesses become the new bottleneck, which diminishes the benefits of batching. As input length and output length increase, throughput decreases due to more KV cache accesses.

From a latency perspective, this throughput-oriented design may not be promising: the latency is 9.21x worse than GA100 on average. While model parameters are only read once for each batch, a larger batch size means more KV cache and intermediate values to read. In LLM inference, there is no free lunch between latency and throughput.

C. Design Space Exploration

With LLMCompass' speed, we are able to conduct design space exploration with different hardware design choices. As shown in Figure 13, four different memory designs with different bandwidths and capacities as well as various core counts and core designs are explored.

Figure 13 indicates our proposed latency-oriented design and throughput-oriented design are around the sweet point. Decreasing compute capability too much can hurt performance due to the compute-intensive *prefill* stage. Increasing memory capacity also has diminishing returns as larger batches increase KV cache accesses.

VI. Discussion

1) ***Hardware designs that can/cannot be modeled by LLMCompass:*** LLMCompass covers the dominant hardware platforms for LLMs today: NVIDIA GPUs, AMD GPUs, and Google TPUs, and can be extended to newer architectures with no/little change to the code, thanks to its generic hardware template and automatic mapping exploration. In LLMCompass, users only need to describe their design and do not need to recalibrate LLMCompass for each new design.

For the three real-world designs evaluated in the paper, we use the same code for performance and area modeling. LLMCompass can be seamlessly extended to newer architectures such as NVIDIA H100. As a train/test setup, we asked our collaborators to validate LLMCompass on an NVIDIA RTX A6000 without changing any code, and LLMCompass achieves within 2.5% error rate for LLM inference workloads.

LLMCompass does not incorporate network modeling, and therefore cannot accurately model Cerebras wafer-scale processors, which have 850K cores and are more like a distributed system where inter-core communication mechanism plays a key role. To model Cerebras-like designs, LLMCompass can add in existing network models [50], [59], [79].

LLMCompass is designed for throughput-oriented latency tolerant machines so it cannot model CPUs accurately due to its latency-sensitive nature and complicated control flow.

2) ***Other optimization techniques:*** LLMCompass can be extended to a variety of optimization techniques. To support operator fusion like FlashAttention [13], users can implement a simulated fused operator based on the simulation code for its individual operators. We do not explore operator fusion in this paper as many of them are specific to NVIDIA GPUs and we are not sure whether they can be applied to other hardware platform such as Google TPUs.

LLMCompass can also be extended to other LLM scheduling techniques. For example, ORCA-style continuous batching [82], SARATHI-style [3] chunked prefills, and Splitwise-style [54] phase splitting can be supported by wrapping a scheduling function on top of LLMCompass.

In this paper, we choose to use request-level batching with different input and output lengths (as in Figure 10, 11, and 12), as it is how NVIDIA benchmarks their TensorRT-LLM [68].

VII. Related Work

A. Evaluating Large-scale Hardware Design

Evaluating the various characteristics of a hardware design, including performance, area, and cost, is extremely useful for hardware designers. To this end, the options are as follows:

Roofline Model Analysis [78]. Roofline models are analytical, fast to evaluate, and can be applied to various architectures for performance comparison. However, they can be overly optimistic relative to actual hardware capabilities.

Cycle-level Simulation [6], [7], [21], [23], [25], [32], [33], [51], [61], [62], [66], [72]. With a typical simulation rate of less than 100K instructions per second, cycle-level simulators become infeasible for design space exploration of LLM scale workloads. As these simulators are often designed for specific architectures, it is hard to describe a hardware design very different from its design purpose (*e.g.*, it's almost impossible to use GPGPU-sim [6] to evaluate a TPU-like design because it relies on the GPU ISA). These simulators often require the user to provide the program for evaluation. If the software program is not optimized, it may lead to unfair comparisons.

FPGA Emulation. Another way is to implement the design in RTL code and emulate it on FPGAs. The RTL code can be either handwritten or generated by accelerator generators [20], [47], [69], [74]. Although the emulation is fast, the synthesis process may take a long time, and users are responsible for mapping their workloads to the hardware. Additionally, users need to repeat this whole process to evaluate a new design.

Comparison. LLMCompass is suitable for pre-silicon design space exploration before diving into more detailed cycle-level simulation or FPGA emulation. Compared with roofline model, LLMCompass is more accurate. As opposed to simulating in a cycle-by-cycle manner, LLMCompass is much faster by leveraging the insights that operators in LLMs follow a highly regular and predictable pattern. Cycle-level simulators are usually tightly bonded to specific architectures. For example, GPGPU-sim [6], [32] only supports a subset of NVIDIA architectures and does not have official support for newer NVIDIA Ampere GPUs like A100. We could not find an existing simulator that models NVIDIA A100, AMD MI210, and Google TPUv3. Compared with FPGA emulation, LLMCompass is significantly less engineering-intensive.

LLMCompass can complement FPGA emulation. Designers can perform initial design space exploration before incurring the heavy costs associated with FPGA emulation and the necessary RTL implementation of the proposed design.

B. Accelerator Design Space Exploration

Since the era of CNN, various works have focused on exploring optimal hardware designs as well as mapping [14], [15], [27], [36], [37], [51], [60], [74], [81], [84]. LLMCompass is different from these works in design considerations and emphasis: ① Mainly targeting Convolutional Neural Networks (CNNs), these works focus on loop parallelization, loop order, and data flows (*e.g.*, weight stationary or output stationary), which are not the primary design considerations in Transformer-based LLMs. LLMCompass is more tailored for matrix multiplication tiling and scheduling as well as other Transformer operators such as *LayerNorm*. ② LLMCompass is designed for GPU-scale designs, which are much larger than CNN accelerators like Eyeriss [10]. LLM workloads are also significantly larger than CNN workloads.

LLMCompass can also complement design space explorations. Implemented as a Python library, LLMCompass can be seamlessly integrated into design space exploration frameworks such as FAST [84]. FAST uses an internal TPU performance simulator, limiting its broader utility. Fast and accurate, we believe the fully open-source LLMCompass can democratize hardware design space exploration research.

C. Accelerating LLM Inference

Many Transformer accelerators have been proposed [26], [67], [75], [76], mainly focusing on accelerating the Transformer with hardware-software co-design such as pruning or approximate-computing. Whether these techniques are effective for the largest of models remains to be seen. Additionally, the major challenge of LLMs today comes from the massive scale of the models, which is the main scope of this paper.

TABLE V: Comparison of Hardware Evaluation Methods

Methods	Fast	Accurate	Architecturally Descriptive◆	Performance Optimal❖	Cost Aware
Roofline	✓	✗	✓	✓	✗
Cycle-level	✗	✓	✗	✻	✗
FPGA	✻	✓	✻	✻	✓
LLMCompass	✓	✓	✓	✓	✓

◆: The ability to describe different hardware designs.
❖: Find the optimal mapping to fully demonstrate hardware capability.

Many efforts have also been made to accelerate LLM inference at the software domain [3], [5], [12], [13], [19], [54], [56], [63], [82]. LLMCompass is compatible with these optimization techniques by modeling their compute and memory access patterns, as discussed in Section VI-2.

VIII. Conclusion

This work introduces LLMCompass, a fast, accurate, and architecturally descriptive hardware evaluation framework for LLM inference workloads. LLMCompass' hardware description template, mapper, and architectural simulator allow hardware designers to evaluate large-scale chip designs for LLMs, which are infeasible for cycle-level simulators. The incorporated area and cost models can also help designers reason about performance-cost trade-offs. With the aid of LLMCompass, we draw implications on how hardware designs affect LLM inference. Based on these findings, we propose a latency-oriented design and a throughput-oriented design that achieve 1.06x and 3.41x performance per cost improvements respectively, compared to NVIDIA GA100. We plan to extend LLMCompass to support more machine learning workloads as well as LLM training/fine-tuning in the future.

Acknowledgements

We would like to thank Qixuan (Maki) Yu, Zhongming Yu, Haiyue Ma, Yanghui Ou, Christopher Batten, and the entire Princeton Parallel Group, for their feedback, suggestions, and encouragement. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-2039656, the National Science Foundation under Grant No. CCF-1822949, Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement No. FA8650-18-2-7862. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

REFERENCES

[1] H. A. Abdelhafez, C. Zimmer, S. S. Vazhkudai, and M. Ripeanu, "Ahead: A tool for projecting next-generation hardware enhancements on gpu-accelerated systems," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 583–592.

[2] Advanced Micro Devices, Inc, "Amd cdna™ 2 architecture," AMD, Tech. Rep., 2021. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf

[3] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills," 2023.

[4] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation," in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, 1995, pp. 95–105.

[5] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He, "Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–15.

[6] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 2009, pp. 163–174.

[7] B. M. Beckmann and A. Gutierrez, "The amd gem5 apu simulator: Modeling heterogeneous systems in gem5," in *Tutorial at the International Symposium on Microarchitecture (MICRO)*, 2015.

[8] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in neural information processing systems*, vol. 33, 2020, pp. 1877–1901.

[9] X. Chen, P. Maniatis, R. Singh, C. Sutton, H. Dai, M. Lin, and D. Zhou, "Spreadsheetcoder: Formula prediction from semi-structured context," in *International Conference on Machine Learning*. PMLR, 2021, pp. 1661–1672.

[10] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, p. 367–379, jun 2016. [Online]. Available: https://doi.org/10.1145/3007787.3001177

[11] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," 2022. [Online]. Available: https://arxiv.org/abs/2204.02311

[12] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023.

[13] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[14] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, "Dmazerunner: Executing perfectly nested loops on dataflow accelerators," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–27, 2019.

[15] S. Dave, A. Shrivastava, Y. Kim, S. Avancha, and K. Lee, "dmazerunner: Optimizing convolutions on dataflow accelerators," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 1544–1548.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[17] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022. [Online]. Available: http://jmlr.org/papers/v23/21-0998.html

[18] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[19] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," 2023.

[20] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 769–774.

[21] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C.-K. Luk, "Performance characterisation and simulation of intel's integrated gpu architecture," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2018, pp. 139–148.

[22] "Github copilot." [Online]. Available: https://github.com/features/copilot

[23] X. Gong, R. Ubal, and D. Kaeli, "Multi2sim kepler: A detailed architectural gpu simulator," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017, pp. 269–278.

[24] "Bard - chat based ai tool from google, powered by palm 2." [Online]. Available: https://bard.google.com/chat

[25] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. G. Rogers, "Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 608–619.

[26] T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, and J. W. Lee, "Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 692–705.

[27] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind mappings: Enabling efficient algorithm-accelerator mapping space search," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 943–958. [Online]. Available: https://doi.org/10.1145/3445814.3446762

[28] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

[29] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre, "Training compute-optimal large language models," 2022.

[30] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Commun. ACM*, vol. 63, no. 7, p. 67–78, jun 2020. [Online]. Available: https://doi.org/10.1145/3360307

[31] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[32] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 473–486.

[33] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.

[34] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *ACM SIGOPS Operating Systems Review*, vol. 25, no. Special Issue, pp. 63–74, 1991.

[35] M. Lapedus, "What's next for high bandwidth memory," *Semiconductor Engineering*, 2019. [Online]. Available: https://semiengineering.com/whats-next-for-high-bandwidth-memory/

[36] R. Li, Y. Xu, A. Sukumaran-Rajam, A. Rountev, and P. Sadayappan, "Analytical characterization and design space exploration for optimization of cnns," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 928–942.

[37] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 553–564.

[38] M. McKeown, A. Lavrov, M. Shahrad, P. J. Jackson, Y. Fu, J. Balkind, T. M. Nguyen, K. Lim, Y. Zhou, and D. Wentzlaff, "Power and energy characterization of an open source 25-core manycore processor," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 762–775.

[39] M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax," *arXiv preprint arXiv:1805.02867*, 2018.

[40] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heintz, and D. Roth, "Recent advances in natural language processing via large pre-trained language models: A survey," *ACM Comput. Surv.*, vol. 56, no. 2, sep 2023. [Online]. Available: https://doi.org/10.1145/3605943

[41] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009.

[42] A. Narayan, I. Chami, L. Orr, S. Arora, and C. Ré, "Can foundation models wrangle your data?" *arXiv preprint arXiv:2205.09911*, 2022.

[43] "nccl-tests." [Online]. Available: https://github.com/NVIDIA/nccl-tests

[44] A. Ning, G. Tziantzioulis, and D. Wentzlaff, "Supply chain aware computer architecture," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589052

[45] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for google's training chips: Tpuv2 and tpuv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, 2021.

[46] "Nvidia ships world's most advanced ai system — nvidia dgx a100 — to fight covid-19; third-generation dgx packs record 5 petaflops of ai performance." [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-ships-worlds-most-advanced-ai-system-nvidia-dgx-a100-to-fight-covid-19-third-generation-dgx-packs-record-5-petaflops-of-ai-performance

[47] NVIDIA Corporation, "The nvidia deep learning accelerator," NVIDIA, Tech. Rep., 2018. [Online]. Available: https://old.hotchips.org/hc30/2conf/2.08_NVidia_DLA_Nvidia_DLA_HotChips_10Aug18.pdf

[48] NVIDIA Corporation, "Nvidia a100 tensor core gpu architecture," NVIDIA, Tech. Rep., 2020. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[49] OpenAI. (2022) Introducing chatgpt. [Online]. Available: https://openai.com/blog/chatgpt

[50] M. Orenes-Vera, E. Tureci, M. Martonosi, and D. Wentzlaff, "Muchisim: A simulation framework for design exploration of multi-chip manycore systems," *arXiv preprint arXiv:2312.10244*, 2023.

[51] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2019, pp. 304–315.

[52] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[53] D. Patel, "Nvidia ada lovelace leaked specifications, die sizes, architecture, cost, and performance analysis," 2022. [Online]. Available: https://www.semianalysis.com/p/nvidia-ada-lovelace-leaked-specifications

[54] P. Patel, E. Choukse, C. Zhang, Íñigo Goiri, A. Shah, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," 2023.

[55] M. Phuong and M. Hutter, "Formal algorithms for transformers," 2022.

[56] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[57] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[58] P. Raghavan, A. Lambrechts, M. Jayapala, F. Catthoor, and D. Verkest, "Empire: Empirical power/area/timing models for register files," *Microprocessors and Microsystems*, vol. 33, no. 4, pp. 295–300, 2009, media and Stream Processing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933109000258

[59] S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 81–92.

[60] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G.-Y. Wei, and D. Brooks, "A case for efficient accelerator design space exploration via bayesian optimization," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2017, pp. 1–6.

[61] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of dnn accelerators using scale-sim," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 58–68.

[62] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[63] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single gpu," in *Proceedings of the 40th International Conference on Machine Learning*, ser. ICML'23. JMLR.org, 2023.

[64] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020.

[65] A. Smith and N. James, "Amd instinct™ mi200 series accelerator and node architectures," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022, pp. 1–23.

[66] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "Mgpusim: Enabling multi-gpu performance modeling and optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 197–209. [Online]. Available: https://doi.org/10.1145/3307650.3322230

[67] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E.-Y. Yang, M. Donato, V. Sanh, P. Whatmough, A. M. Rush, D. Brooks, and G.-Y. Wei, "Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 830–844. [Online]. Available: https://doi.org/10.1145/3466752.3480095

[68] "Tensorrt-llm." [Online]. Available: https://github.com/NVIDIA/TensorRT-LLM/blob/main/docs/source/performance.md

[69] B. Tine, V. Saxena, S. Srivatsan, J. R. Simpson, F. Alzammar, L. Cooper, and H. Kim, "Skybox: Open-source graphic rendering on programmable risc-v gpus," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 616–630. [Online]. Available: https://doi.org/10.1145/3582016.3582024

[70] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[71] TrendForce, "Dram spot price," 2023. [Online]. Available: https://www.dramexchange.com/

[72] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 335–344.

[73] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[74] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer,

W. J. Dally, J. Emer, S. W. Keckler, and B. Khailany, "Magnet: A modular accelerator generator for neural networks," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[75] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 97–110.

[76] Y. Wang, Y. Qin, D. Deng, J. Wei, Y. Zhou, Y. Fan, T. Chen, H. Sun, L. Liu, S. Wei, and S. Yin, "A 28nm 27.5tops/w approximate-computing-based transformer processor with asymptotic sparsity speculating and out-of-order computing," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 1–3.

[77] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," 2022. [Online]. Available: https://arxiv.org/abs/2206.07682

[78] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[79] W. Won, T. Heo, S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "Astra-sim2. 0: Modeling hierarchical networks and disaggregated systems for large-model training at scale," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 283–294.

[80] B. Workshop, T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, J. Tow, A. M. Rush, S. Biderman, A. Webson, P. S. Ammanamanchi, T. Wang, B. Sagot, N. Muennighoff, A. V. del Moral, O. Ruwase, R. Bawden, S. Bekman, A. McMillan-Major, I. Beltagy, H. Nguyen, L. Saulnier, S. Tan, P. O. Suarez, V. Sanh, H. Laurençon, Y. Jernite, J. Launay, M. Mitchell, C. Raffel, A. Gokaslan, A. Simhi, A. Soroa, A. F. Aji, A. Alfassy, A. Rogers, A. K. Nitzav, C. Xu, C. Mou, C. Emezue, C. Klamm, C. Leong, D. van Strien, D. I. Adelani, D. Radev, E. G. Ponferrada, E. Levkovizh, E. Kim, E. B. Natan, F. D. Toni, G. Dupont, G. Kruszewski, G. Pistilli, H. Elsahar, H. Benyamina, H. Tran, I. Yu, I. Abdulmumin, I. Johnson, I. Gonzalez-Dios, J. de la Rosa, J. Chim, J. Dodge, J. Zhu, J. Chang, J. Frohberg, J. Tobing, J. Bhattacharjee, K. Almubarak, K. Chen, K. Lo, L. V. Werra, L. Weber, L. Phan, L. B. allal, L. Tanguy, M. Dey, M. R. Muñoz, M. Masoud, M. Grandury, M. Šaško, M. Huang, M. Coavoux, M. Singh, M. T.-J. Jiang, M. C. Vu, M. A. Jauhar, M. Ghaleb, N. Subramani, N. Kassner, N. Khamis, O. Nguyen, O. Espejel, O. de Gibert, P. Villegas, P. Henderson, P. Colombo, P. Amuok, Q. Lhoest, R. Harliman, R. Bommasani, R. L. López, R. Ribeiro, S. Osei, S. Pyysalo, S. Nagel, S. Bose, S. H. Muhammad, S. Sharma, S. Longpre, S. Nikpoor, S. Silberberg, S. Pai, S. Zink, T. T. Torrent, T. Schick, T. Thrush, V. Danchev, V. Nikoulina, V. Laippala, V. Lepercq, V. Prabhu, Z. Alyafeai, Z. Talat, A. Raja, B. Heinzerling, C. Si, D. E. Taşar, E. Salesky, S. J. Mielke, W. Y. Lee, A. Sharma, A. Santilli, A. Chaffin, A. Stiegler, D. Datta, E. Szczechla, G. Chhablani, H. Wang, H. Pandey, H. Strobelt, J. A. Fries, J. Rozen, L. Gao, L. Sutawika, M. S. Bari, M. S. Al-shaibani, M. Manica, N. Nayak, R. Teehan, S. Albanie, S. Shen, S. Ben-David, S. H. Bach, T. Kim, T. Bers, T. Fevry, T. Neeraj, U. Thakker, V. Raunak, X. Tang, Z.-X. Yong, Z. Sun, S. Brody, Y. Uri, H. Tojarieh, A. Roberts, H. W. Chung, J. Tae, J. Phang, O. Press, C. Li, D. Narayanan, H. Bourfoune, J. Casper, J. Rasley, M. Ryabinin, M. Mishra, M. Zhang, M. Shoeybi, M. Peyrounette, N. Patry, N. Tazi, O. Sanseviero, P. von Platen, P. Cornette, P. F. Lavallée, R. Lacroix, S. Rajbhandari, S. Gandhi, S. Smith, S. Requena, S. Patil, T. Dettmers, A. Baruwa, A. Singh, A. Cheveleva, A.-L. Ligozat, A. Subramonian, A. Névéol, C. Lovering, D. Garrette, D. Tunuguntla, E. Reiter, E. Taktasheva, E. Voloshina, E. Bogdanov, G. I. Winata, H. Schoelkopf, J.-C. Kalo, J. Novikova, J. Z. Forde, J. Clive, J. Kasai, K. Kawamura, L. Hazan, M. Carpuat, M. Clinciu, N. Kim, N. Cheng, O. Serikov, O. Antverg, O. van der Wal, R. Zhang, R. Zhang, S. Gehrmann, S. Mirkin, S. Pais, T. Shavrina, T. Scialom, T. Yun, T. Limisiewicz, V. Rieser, V. Protasov, V. Mikhailov, Y. Pruksachatkun, Y. Belinkov, Z. Bamberger, Z. Kasner, A. Rueda, A. Pestana, A. Feizpour, A. Khan, A. Faranak, A. Santos, A. Hevia, A. Unldreaj, A. Aghagol, A. Abdollahi, A. Tammour, A. HajiHosseini, B. Behroozi, B. Ajibade, B. Saxena, C. M. Ferrandis, D. McDuff, D. Contractor, D. Lansky, D. David, D. Kiela, D. A. Nguyen, E. Tan, E. Baylor, E. Ozoani, F. Mirza, F. Ononiwu, H. Rezanejad, H. Jones, I. Bhattacharya, I. Solaiman, I. Sedenko, I. Nejadgholi, J. Passmore, J. Seltzer, J. B. Sanz, L. Dutra, M. Samagaio, M. Elbadri, M. Mieskes, M. Gerchick, M. Akinlolu, M. McKenna, M. Qiu, M. Ghauri, M. Burynok, N. Abrar, N. Rajani, N. Elkott, N. Fahmy, O. Samuel, R. An, R. Kromann, R. Hao, S. Alizadeh, S. Shubber, S. Wang, S. Roy, S. Viguier, T. Le, T. Oyebade, T. Le, Y. Yang, Z. Nguyen, A. R. Kashyap, A. Palasciano, A. Callahan, A. Shukla, A. Miranda-Escalada, A. Singh, B. Beilharz, B. Wang, C. Brito, C. Zhou, C. Jain, C. Xu, C. Fourrier, D. L. Periñán, D. Molano, D. Yu, E. Manjavacas, F. Barth, F. Fuhrimann, G. Altay, G. Bayrak, G. Burns, H. U. Vrabec, I. Bello, I. Dash, J. Kang, J. Giorgi, J. Golde, J. D. Posada, K. R. Sivaraman, L. Bulchandani, L. Liu, L. Shinzato, M. H. de Bykhovetz, M. Takeuchi, M. Pàmies, M. A. Castillo, M. Nezhurina, M. Sänger, M. Samwald, M. Cullan, M. Weinberg, M. D. Wolf, M. Mihaljcic, M. Liu, M. Freidank, M. Kang, N. Seelam, N. Dahlberg, N. M. Broad, N. Muellner, P. Fung, P. Haller, R. Chandrasekhar, R. Eisenberg, R. Martin, R. Canalli, R. Su, R. Su, S. Cahyawijaya, S. Garda, S. S. Deshmukh, S. Mishra, S. Kiblawi, S. Ott, S. Sang-aroonsiri, S. Kumar, S. Schweter, S. Bharati, T. Laud, T. Gigant, T. Kainuma, W. Kusa, Y. Labrak, Y. S. Bajaj, Y. Venkatraman, Y. Xu, Y. Xu, Y. Xu, Z. Tan, Z. Xie, Z. Ye, M. Bras, Y. Belkada, and T. Wolf, "Bloom: A 176b-parameter open-access multilingual language model," 2023.

[81] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 369–383. [Online]. Available: https://doi.org/10.1145/3373376.3378514

[82] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 521–538.

[83] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[84] D. Zhang, S. Huda, E. Songhori, K. Prabhu, Q. Le, A. Goldie, and A. Mirhoseini, "A full-stack search technique for domain optimized deep learning accelerators," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 27–42.

ARTIFACT APPENDIX

A. Abstract

This artifact appendix illustrates how to utilize the proposed LLMCompass framework to evaluate hardware designs running Large Language Model (LLM) workloads. An x86 machine capable of running Python programs is required to reproduce all the results in Figure 5-12.

B. Artifact Checklist

- **Run-time environment:** Python 3.9
- **Hardware:** An x86 machine.
- **Metrics:** Throughput and latency of serving LLM inference. Area and cost of the hardware design.
- **Output:** CSV logs and Figure 5-12 in the paper.
- **Experiments:** Scripts are provided to automate the experimental flow and generate the graphs.
- **How much disk space required (approximately)?:** 1GB.
- **How much time is needed to prepare workflow (approximately)?:** Less than an hour.
- **How much time is needed to complete experiments (approximately)?:** 10 hours.
- **Publicly available?:** On GitHub: https://github.com/PrincetonUniversity/LLMCompass.
- **Archived?:** https://doi.org/10.5281/zenodo.10951545.

C. Description

1) How to access: The artifact is available at https://github.com/PrincetonUniversity/LLMCompass and includes all the source code, scripts, and data that are sufficient to reproduce all the experiments in the paper.

2) Hardware dependencies: The artifact works on x86 CPUs and has been verified on an Intel Xeon Gold 6242R CPU @ 3.10GHz.

3) Software dependencies: The artifact requires Python3 and has been verified with Python 3.9.19. We recommend using Anaconda to create a virtual environment and install the required packages as below:

```
$ conda create -n llmcompass_ae python=3.9
$ conda activate llmcompass_ae
$ pip3 install scalesim
$ conda install pytorch==2.0.0 -c pytorch
$ pip3 install matplotlib
$ pip3 install seaborn
$ pip3 install scipy
```

4) Installation: We use Python module method so no installation required. One can download the artifact as below:

```
$ git clone -b ISCA_AE https://github.com/
PrincetonUniversity/LLMCompass
$ cd LLMCompass
$ git submodule init
$ git submodule update --recursive
```

D. Experiment workflow

After setting up the environment, one can reproduce the experiments by running the following scripts. These scripts will first generate CSV files and then visualize the results as in the paper. There is no dependency on these scripts and feel free to run them in parallel.

```
# Figure 5 (around 100 min)
$ cd ae/figure5
$ bash run_figure5.sh

# Figure 6 (around 1 min)
$ cd ae/figure6
$ bash run_figure6.sh

# Figure 7 (around 20 min)
$ cd ae/figure7
$ bash run_figure7.sh

# Figure 8 (around 40 min)
$ cd ae/figure8
$ bash run_figure8.sh

# Figure 9 (around 30 min)
$ cd ae/figure9
$ bash run_figure9.sh

# Figure 10 (around 45 min)
$ cd ae/figure10
$ bash run_figure10.sh

# Figure 11 (around 5 min)
$ cd ae/figure11
$ bash run_figure11.sh

# Figure 12 (around 4 hours)
$ cd ae/figure12
$ bash run_figure12.sh
```

Profiling results on real world hardware has been provided in advance (as illustrated in Sec. III-C) and is out of the scope of this artifact.

E. Evaluation and expected result

After running each script above, the corresponding figures will be generated under the corresponding directory as suggested by its name:

- Figure 5a and Figure 5b: `ae\figure5\ab`
- Figure 5c and Figure 5f: `ae\figure5\cf`
- Figure 5d and Figure 5e: `ae\figure5\de`
- Figure 5g: `ae\figure5\g`
- Figure 5h: `ae\figure5\h`
- Figure 5i-l: `ae\figure5\ijkl`
- Figure 6: `ae\figure6`
- Figure 7: `ae\figure7`
- Figure 8: `ae\figure8`
- Figure 9: `ae\figure9`
- Figure 10: `ae\figure10`
- Figure 11: `ae\figure11`
- Figure 12: `ae\figure12`

For comparison, a copy of the expected results can be found in `ae\expected_results`. The figures generated by the scripts should be identical to these expected results. Minor mismatch is possible because we are actively improving the framework after the paper submission. However, the mismatch should not exceed 5%.

The only exception is Figure 11, which has been replaced by a new figure to show the TTFT and TBT of the proposed latency-oriented design. A copy of the original Figure 11 can be found in `ae\expected_results`.

F. Experiment customization

1) Customized hardware design: In addition to the provided hardware configurations, users can describe their own hardware design with the provided hardware description template `configs\template.json`. More examples can be found in `configs\` and users need to bring their own numbers and set the parameters in the `json` file.

An alternative way is to build up the hardware design bottom-up with the provided Python Class defined in `hardware_model\`. The user needs to define their own `ComputeModule`, `MemoryModule`, `IOModule`, and `InterConnectModule`, and combine them into a `System`.

2) Customized LLM computational graph: In addition to the provided Multi-Head-Attention Transformer, users can describe their own computational graph with the provided operators and primitives, including `Matmul`, `LayerNorm`, `Softmax`, `GeLU`, and `AllReduce`. An example is shown in `software_model\transformer`. The user needs to initialize the operators and combine them into a computational graph in a similar way to PyTorch.

G. Extending LLMCompass

*1) **Other DNN models such as RNN or CNN**:* LLMCompass focuses on Transformers as almost all the LLMs today are based on Transformers. LLMCompass can also support other DNN models as long as they can be expressed by the operators LLMCompass has. For RNNs such as LSTM, LLMCompass already supports all the operators and we can model the recurrent nature of RNNs similar to the auto-regressive decoding stage of LLMs. To support CNNs, we can derive the convolution operator by modifying the existing matrix multiplication code.

*2) **Other precisions**:* LLMCompass naturally supports other precisions by allowing users to define their own data type. In this work, we usually use FP16/BF16 and FP32 because they are widely used and naturally supported by the GPUs/TPUs we have. LLMCompass takes the data type as an input and different precisions will consume different bandwidth and compute operations. LLMCompass can help users explore the different speeds of different precisions and facilitate low-precision LLM research, such as GPTQ [19]. LLMCompass cannot explore the accuracy of quantized models as it does not perform numerical computations.

*3) **Training and fine-tuning**:* LLMCompass can be extended to fine-tuning by building the back propagation computational graph with the provided operator. This generally does not require implementing new operators as the current operators are sufficient for LLMs (for instance, the backward pass of a matrix multiplication is also matrix multiplications). Optimizer and weight update can be modeled as element-wise operations which are already supported by LLMCompass.

Training would require a network model due to the large scale of hardware systems involved to simulate the communication and synchronization among different nodes. This can be done by integrating LLMCompass with a network simulator such as ASTRA-sim [59], [79], where LLMCompass serves as a truthful device-level hardware backend.