SplitQuant: Resource-Efficient LLM Offline Serving on Heterogeneous GPUs via Phase-Aware Model Partition and Adaptive Quantization

Juntao Zhao[1], Borui Wan[1], Yanghua Peng[2], Haibin Lin[2], Chuan Wu[1]
[1]The University of Hong Kong, Hong Kong, [2] ByteDance Inc., USA
juntaozh, wanborui@connect.hku.hk, cwu@cs.hku.hk, pengyanghua.yanghua, haibin.lin@bytedance.com

***Abstract*—Modern large language models (LLMs) serving systems address distributed deployment challenges through two key techniques: distributed model partitioning for parallel computation across accelerators and quantization for reducing parameter size. While existing systems assume homogeneous GPU environments, we reveal significant untapped potential in heterogeneous systems with mixed-capacity accelerators where two critical limitations persist: (1) uniform partitioning and quantization strategies fail to adapt to hardware heterogeneity, exacerbating resource imbalance, and (2) decoupled optimization of partitioning and quantization overlooks critical performance synergies between these techniques. We present SplitQuant, a phase-aware distributed serving system that co-optimizes mixed-precision quantization, phase-aware model partitioning, and micro-batch sizing for heterogeneous environments. Our approach combines analytical modeling of quality-runtime tradeoffs with a lightweight planning algorithm to maximize throughput while preserving user-specified model quality targets. Evaluations across 10 production clusters show SplitQuant achieves up to 2.34× (1.61× mean) higher throughput than state-of-the-art approaches without violating accuracy targets. Our results underscore the value of co-designing quantization and model partitioning strategies for heterogeneous environments.**

***Index Terms*—Large Language Models (LLMs), Inference, Heterogeneous Computing, Model Partitioning, Quantization**

I. Introduction

Large language models (LLMs) [1]–[4] have demonstrated unprecedented capabilities across diverse tasks. The outstanding model performance is largely attributed to a very large model size ranging from a few hundred million to even half a trillion parameters. Serving a trained LLM is also resource-demanding and cost-intensive, as common LLMs cannot fit into a single GPU, therefore, multiple GPUs are required for distributed inference.

To cope with the massive size of LLMs, model partition and compression techniques have been proposed to enable their efficient deployment in practice. Contemporary frameworks such as vLLM [5], SGLang [6], and TensorRT-LLM [7] leverage tensor parallelism (TP) [8] and pipeline parallelism (PP) [9] to perform model partition at intra- and inter-operator granularity, coupled with quantization as main compression schemes [10], [11], converting weights to lower-precision formats (e.g., 8-bit integers) to reduce the memory footprint. However, the existing solutions are mainly designed for models serving on homogeneous clusters, limiting their performance in a heterogeneous cluster.

While newly built machine learning (ML) clusters typically employ homogeneous GPUs, production environments frequently exhibit hardware heterogeneity, incorporating GPU models from different generations that were purchased at different times. Utilization of different types of GPUs may differ substantially. Fig. 1 shows the proportion of different GPUs in a production cluster, with fewer percentages of high-calibre GPUs (NVIDIA A100) the majority being relatively low-calibre inference GPUs (such as T4, V100). The utilization rate of other GPUs is much lower than that of A100, which is used intensively for both training and inference of large models nowadays, for the best performance. **Efficiently exploiting available heterogeneous GPUs for LLM serving** is worthwhile to explore, to fully utilize available resources and reduce the cost of provisioning LLM-enabled applications.

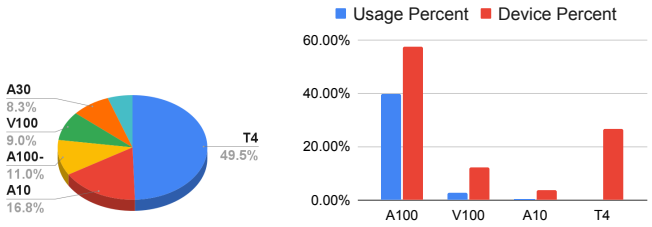


Fig. 1: Statistics from our production cluster. (a) Distribution of GPU types in the cluster. (b) Monthly average utilization rates per GPU type. Usage percent is effective GPU hours divided by total GPU available hours.

Existing PP paradigms evenly partition model operations across homogeneous GPUs, leading to suboptimal resource utilization in heterogeneous clusters: High-capacity GPUs remain underutilized while low-memory devices risk out-of-memory (OOM) errors. The prior studies of models serving on heterogeneous clusters [12] focus on the partition of encoder-based transformer models. However, mainstream LLMs with decoder-only structures contain two phases during inference: prompt processing (prefill) and token generation (decode). While the former phase is similar to the inference of encoder-based transformers, the latter has a totally different pattern (see Sec. II-A), making the previous partition solutions not suitable. Crucially, the relative duration of these phases varies dynamically based on prompt length and output token count, a variance further exacerbated in heterogeneous clusters. Existing approaches that optimize solely for the prefill phase

thus yield poor end-to-end performance. Additionally, extra memory required for pre- and post-processing during LLM inference, such as text embedding for converting input tokens to word vectors, should also be considered, especially when utilizing GPUs with limited memory.

When the model is partitioned among heterogeneous GPUs, adopting a uniform quantization precision across all model layers in different types of GPUs is always suboptimal. Uniform quantization strategy can select a precision, e.g., INT4, that is suitable for GPUs with lower memory to avoid the OOM (out of memory) problem, but causes a notable portion of memory underutilization for those with abundant GPU memory. Adaptive mixed-precision quantization for LLM, which is not investigated in the literature [11], [13], is more desirable. By using higher precision for model weights on GPUs with more available memory instead of forcing them to use the same one in those low-calibre GPUs, adaptive mixed-precision quantization can not only avoid memory underutilization but also promote the model quality as well.

Quantization-aware partitioning introduces additional system challenges due to hardware heterogeneity. Variations in arithmetic intensity (FLOPs/Bytes) and hardware support for quantization primitives (e.g., tensor cores) mean even identical bitwidths yield divergent performance across devices. Furthermore, layers exhibit varying quantization sensitivity, requiring rigorous modeling of the accuracy-performance tradeoff to maximize throughput under user-specified quality constraints.

In this work, we present SplitQuant, a novel system for efficient LLM generative serving on heterogeneous GPU clusters. When using low-capability GPUs that may struggle to meet real-time SLOs, SplitQuant focuses on efficient processing of offline serving workloads. It adopts adaptive model quantization, phase-aware model partition, and efficient micro-batch scheduling for LLM pipeline serving. Given the LLM, workload profile, cluster resources, and user-specified quality targets, it jointly decides on quantization precisions, model layer partition, and micro-batch sizing. Our contributions can be summarized as follows:

▷ We provide a cost model that details the memory requirements of LLM serving under a mixed-precision quantization scheme. We learn a linear regression model to accurately predict the latency of mixed-precision LLM inference workloads with varying sequence lengths and batch sizes based on their phase-aware computational characteristics.

▷ We introduce adaptive mixed precision in the search space for the heterogeneous LLM pipeline serving and provide a variance indicator to measure the sensitivity of the layer towards the quantization. We develop an iterative algorithm that first explores possible GPU orderings and different (phase, micro-batch size) pairs in the pruned search space, and then solves an integer linear programming (ILP) problem to determine the best partition and quantization bitwidths.

▷ We present a prototype of SplitQuant. To validate its efficacy, we experiment across 10 heterogeneous GPU clusters comprising prevalent architectures (T4, P100, V100, A100). Our evaluation shows that SplitQuant achieves up to 2.34× throughput improvement (1.61× on average) compared to the latest approaches. SplitQuant is a general-purpose solution that can be adapted to other accelerators [14].

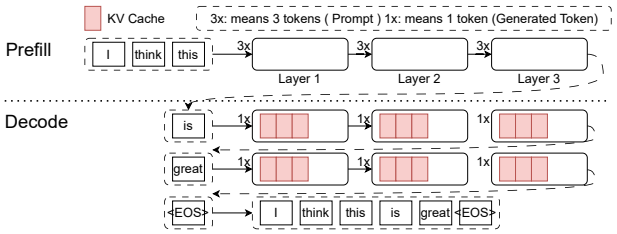


Fig. 2: Two phases in LLM generative serving: (Top) **Prefill phase** takes the prompt sequence to generate the initial key-value pairs. (Bottom) **Decode phase** takes previously generated token & stored KV pairs to generate the next token.

II. Background and Motivation

A. Generative Inference of LLM

LLM generally refers to a suite of decoder-only transformer models with large parameter sizes [2], [3]. Unlike encoder-based transformers like ViT-Huge [15] and Bert-Large [16] that are sequence-to-sequence, LLMs generate tokens one by one in an inference process that comprises two phases [2], [3](Fig. 2): **prefill** and **decode** [17]. In the prefill phase, the entire input prompt is processed to produce an initial key/value (KV) cache. The decode phase then autoregressively generates one token at a time, using the updated KV cache from previous steps. In token generation, each layer of the LLM undergoes a prefill phase followed by several passes in the decode phase (an example is given in Fig. 2).

The time taken by the prefill and decode phases varies to the prompt length and output token length. By sampling 10,000 conversations from the ShareGPT [18] dataset, we found that the prompt length varies substantially: < 128 (14.20%), 129-512 (20.52%), 513-1024 (14.24%), 1025-2048 (14.53%) and others (36.51%). In the upper part of Fig. 3, we evaluate the time required to process a batch of 8 sequences and generate 32 tokens per sequence, with prompt lengths of 1024 and 128 on OPT-13B and OPT-30B models [3], respectively. The prefill time increases with the prompt length (as it processes all the prompt tokens once) and is substantial ($\geq 36\%$) when the prompt is long. Unlike prefill time, the decode time is determined by the number of generated tokens. These characteristics make the inference pattern of LLM more complicated than encoder-based transformers.

B. Heterogenous Model Parallelization

Tensor Parallelism (TP) [8] and Pipeline Parallelism (PP) [9], [19] are widely used to distribute LLM parameters across devices. TP partitions model weights, while PP splits the model into stages and processes micro-batches in a pipelined manner. TP has high communication overhead, making it bandwidth-sensitive. In our work, we force intra-node TP to avoid intensive cross-node communication and primarily focus on analyzing PP. Workload balance across PP stages is essential, as the slowest stage limits the overall pipeline throughput.

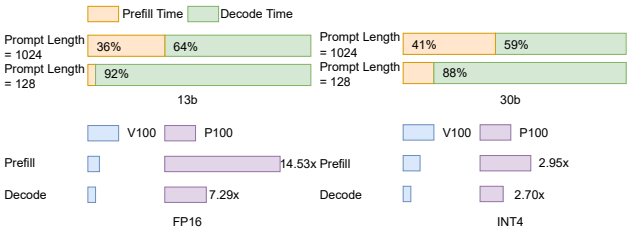
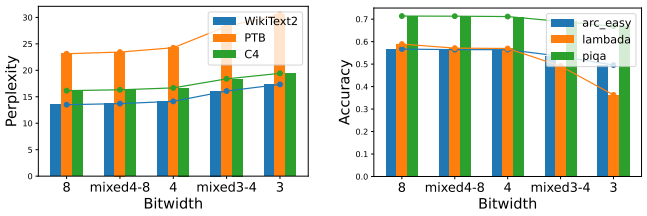


Fig. 3: Phase time decomposition with different precisions. × indicates time on P100 compared to V100.



(a) (b)

Fig. 4: BLOOM-3B (a) and OPT-1.3B (b) perplexity & accuracy under different quantization schemes. Smaller PPL means the model is more confident in its prediction.

Deriving optimal partitions among heterogeneous devices is challenging, especially when considering the two-phase token generation. The lower part of Fig. 3 gives the execution time of a single layer of the respective model with prompt length 512 and batch size 8. The execution time ratio when running the same phase on different devices varies substantially. For example, under FP16, the execution time of the layer in the prefill phase on P100 is 14.53× larger than that on V100, while the execution time ratio is 7.29× for the decode phase. Since the LLM inference time contains these two phases, pipeline stage partitioning should consider the execution time of both phases on each GPU.

Moreover, a full-fledged language model comprises an embedding layer and an LM-head layer, which handle the conversion between sentences and word vectors. In heterogeneous GPU clusters, these layers can exacerbate memory and computational imbalances due to the diverse computing and memory capabilities of different GPU models.

C. Online and Offline Serving

Modern LLM inference workloads fall into two broad categories. **Online serving** processes real-time user requests with variable prompt lengths and token generation counts. Examples include chatbots, code assistants, and interactive applications [1]. In this context, token generation latency is critical, necessitating strict adherence to service-level objective (SLO) requirements such as Time to First Token (TTFT) and Time Between Tokens (TBT) [5], [20], [21]. **Offline serving**, in contrast, comprises batched prompt processing optimized for throughput rather than latency. Common use cases include document summarization [22], synthetic data generation, model checkpoint evaluation [23], and long-context understanding tasks [24]. Offline serving workloads are commonly predictable: dedicated servers handle specific tasks, enabling practitioners to profile and optimize workloads in advance.

Lower-tier heterogeneous GPU clusters often struggle to meet real-time serving demands. SplitQuant therefore specializes in **offline serving** scenarios where prompt length distributions and token generation requirements can be predefined or statistically predicted.

Opportunity 1: Phase-Aware Model Partition on Heterogeneous GPUs. By jointly modeling prefill and decode-phase execution patterns while accounting for resource demands from embedding and LM-head layers, we can establish a comprehensive workload characterization. This characterization enables phase-aware model partitioning that achieves significantly improved performance in offline serving deployment over heterogeneous GPU environments.

D. Quantization in LLM

Quantization is a model compression technique that maps high-precision values, such as those stored in FP16, to their low-precision counterparts. For symmetric quantization, the input data or model weight distribution is evenly partitioned into a fixed number of bins. Each bin is rounded to an n-bit quantized value using $\hat{x} = \lfloor \frac{x - q_x}{s_x} \rceil$, where x is the original value in floating-point format, q_x and s_x are the zero-point and scaling factor, respectively, $\lfloor \cdot \rceil$ is the rounding function and $\hat{x}$ is the resulting quantized value in lower-precision form. For each element $x \in$ vector $\mathbf{x}$, the scaling factor is derived as $s_x = \frac{\mathbf{x}_{max} - \mathbf{x}_{min}}{2^b - 1}$, where $\mathbf{x}_{max}$ and $\mathbf{x}_{min}$ are maximum and minimum values of the vector, and b is the bitwidth. Dequantization is done with $\tilde{x} = s_x \hat{x} + q_x$, where $\tilde{x}$ is the dequantized value in floating point.

LLM Quantization Schemes. LLMs typically store weights in FP16/BF16 formats. The colossal memory footprint of these weights necessitates further compression for efficient serving. For instance, INT8 quantization halves weight storage by reducing precision. Standard LLM quantization approaches fall into two categories: (1) Weight-Activation quantization (e.g., SmoothQuant [13] and ZeroQuant [25]), which quantizes both weights and activations during inference (e.g., W4A4, W4A8, W8A8). (2) weight-only quantization, (e.g., GPTQ [11], AWQ [10]), where only weights are quantized when loaded into GPU memory (e.g., W8A16, W4A16). In this work, we adopt weight-activation quantization using bitsandbytes [26] and SmoothQuant [13]. For weight-only quantization, we leverage GPTQ [11]'s optimized kernels. We exclude discussion of configurations like W4A4 and W4A8 due to their lack of practical speedups over W4A16 and insufficient accuracy in integrated serving platforms such as vLLM [5] and SGLang [6].

A limitation of existing LLM quantization works is that they uniformly quantize all model layers to the same bit precision by default (e.g., 3, 4, or 8 bits [11], [26]). This approach results in either underutilized memory on high-end GPUs or out-of-memory (OOM) issues on low-end GPUs within a heterogeneous cluster. The root cause is that different types of GPUs are unable to select the most appropriate precisions to align with their diverse capabilities.

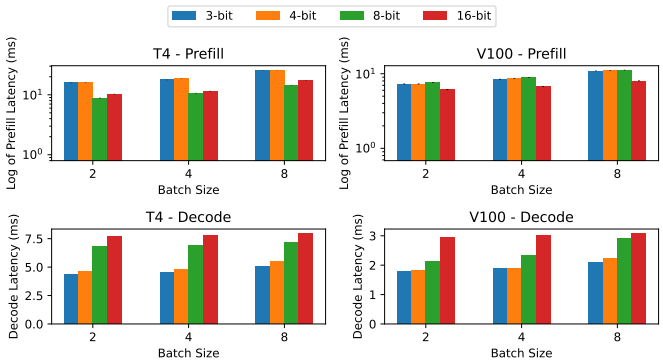


Fig. 5: Execution time of prefill and decode phases under different precisions and batch sizes.

Opportunity 2: Adaptive Quantization for Better Accuracy and Speed. We advocate adaptive quantization by choosing potentially different bitwidths for model layers on different GPUs, to better utilize the available memory, as well as to improve model quality and computation speed as compared to uniform quantization. We illustrate the benefits of adaptive quantization as follows:

1. Adaptive quantization can lead to better model accuracy. We run BLOOM-3B1 [2] and OPT-1.3B with different precision setups on A100 and evaluate the *perplexity* [27], on three text datasets [28]–[30]. PPL measures how well the model predicts the next word in a sequence, the lower the better. We also measure the model *accuracy* on popular zero-shot question-answering benchmarks LAMBADA [31], ARC [32] and PIQA [33]; We use calibration data from the C4 dataset to determine quantization statistics. In Fig. 4, 'mixed4-8' and 'mixed3-4' denote per-layer stochastic quantization strategies employing random bit-width allocations of $\{4, 8\}$ and $\{3, 4\}$, respectively. Results show mixed-precision quantization preserves model accuracy more effectively than uniform low-bit approaches by strategically allocating higher bit-widths to minimize critical weight compression.

2. Adaptive quantization speeds up inference. The decode phase of LLM inference is predominantly memory-bound, exacerbated by the widening compute-to-memory gap in modern GPUs (e.g., T4, A100 200× FLOPs/Bytes intensity). Quantization alleviates this bottleneck by reducing the volume of data movement, as validated by prior work [34]. However, the performance impact of quantization represents a trade-off: while lower precision reduces memory traffic, it increases compute overheads due to dequantization and arithmetic operations. Consequently, the net speedup depends on device capabilities (e.g., bandwidth and FLOPs) and workload characteristics (i.e., input shape).

Fig. 5 shows how quantization performs with different device types and input shapes. The latency is measured on a single layer of OPT-30B with a prompt length of 512. We observe that uniform low-precision quantization may not always result in inference speed-up. Crucially, FP16 precision retains computational advantages during prefill phases versus low precisions (3,4 bit). When uniform quantization (e.g., to INT8) leaves residual GPU memory capacity, selectively replacing layers with high-precision kernels (FP16) can optimize long-context processing. For example, deploying INT8-FP16 hybrid computation on V100 in such memory semi-abundant scenarios reduces operational overhead.

TABLE I: Model performance comparison under different layer quantizations. The best results are marked in bold. Unselected layers are retained in FP16.

Model	Layers Quantized to 4-bit	Avg. Perplexity	Avg. Accuracy (%)
OPT-1.3b	0-8	**15.52**	**62.82**
	8-16	15.78	62.49
	16-24	15.98	61.67
BLOOM-3b	0-10	**17.65**	**60.71**
	10-20	17.88	60.24
	20-30	17.94	60.37

E. Challenges

Adopting adaptive mixed-precision with an asymmetrically partitioned model poses new challenges. Quantization bitwidth (precision) selection must be considered jointly with layer partition, as the same quantized kernel can perform differently on different GPUs due to their precision support and vary FLOPs/Bytes intensity, as shown in Fig. 3 and Fig. 5. For example, T4 supports fast INT8 due to its tensor core, making the execution time of the 8-bit layer comparable to FP16, while V100's INT8 performance depends on the input shape. Other factors, such as micro-batch size, prompt length, and token generation number, also affect the kernel speed and pipeline bubbles in the prefill and decode phases. To produce an optimized inference execution plan, we should take into account all these factors, which results in a complex problem with a huge solution space.

First, determining the optimal inference execution plan requires an accurate estimation of memory and latency across devices under different precisions. Profiling every possible combination of precision, GPU type, and input shape for all partition cases would be very time-consuming. An efficient cost model is needed to reduce the overhead. *Second*, different layers in an LLM may exhibit different sensitivities to quantization, in terms of model performance impact, when quantized to the same bit. Table I shows that selecting different layers of LLMs for quantization can render different model qualities. This finding highlights the importance of identifying a suitable layer quantization sensitivity indicator to guide bits selection, achieving the goal of reducing memory waste and promoting model quality simultaneously. *Last*, due to the large solution space of our joint decision-making problem, offline search for optimal solutions can still be time-consuming. An efficient algorithm is in need to effectively prune the solution space.

We design SplitQuant to handle all these challenges and achieve significant performance gains of LLM serving on heterogeneous clusters.

III. SplitQuant Overview

SplitQuant comprises an offline assigner and a distributed model inference runtime, as illustrated in Fig. 6. The system operates in three phases: ❶ **Input Configuration**: Users specify (i) the pre-trained LLM architecture, (ii) device resource configurations in the heterogeneous cluster, (iii) supported quantization precisions, (iv) a query workload profile

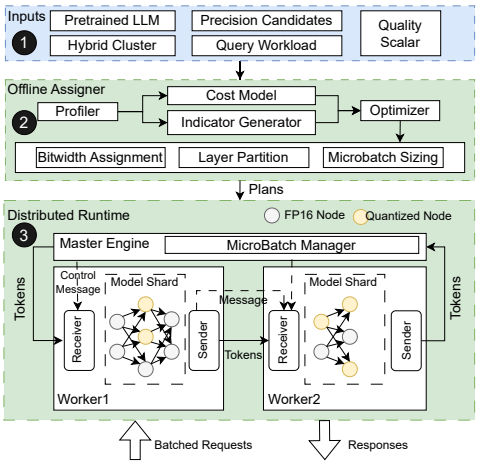


Fig. 6: SplitQuant Overview

TABLE II: Notation

h_1	Hidden dimension of Transformer layers	h_2	Hidden dimension of 2nd MLP layer
v	Batch size	s	Prompt length
t	Index of current generated token	bit	Bitwidth of the current layer
d_t	Dimension of word embedding projection	d_p	Dimension of position embedding
$vocab_s$	Vocabulary size	pos_s	Max position embeddings

(including prompt/output length distributions, and maximum request counts), and (v) a *quality scalar* to explicitly balance model quality against resource efficiency (Sec. IV-C). ❷ **Offline Optimization**: The assigner profiles memory/latency characteristics across devices using GPU calibration payloads to fit the cost model. Simultaneously, the Indicator Generator quantifies per-layer accuracy degradation under varying bit-widths. These inputs drive the optimizer to determine: (i) per-layer quantization bits, (ii) cross-device layer partitioning, and (iii) pipeline micro-batch sizes. ❸ **Distributed Execution**: The runtime executes generation inference via the optimizer's plan. A centralized master engine handles preprocessing (token embeddings) and postprocessing (logit-to-token conversion), while dynamically adapting micro-batch sizes across generation phases. Each worker process manages one pipeline stage spanning one or multiple (TP) dedicated GPUs.

IV. Assigner Design

A. Cost Model

Memory Cost Model. Memory is a first-class citizen in LLM serving systems. The peak memory usage of pipeline LLM serving is largely due to the model weights, the KV cache for all requests, and the peak activation storage required by the model layers across different inference phases.

For *weights*, we consider embedding weights (token embeddings of size $vocab_s \times d_t$, position embeddings of size $pos_s \times d_t$, and projections of size $2 \times h_1 \times d_t$ when $h_1 \neq d_t$), the LM head of size $vocab_s \times d_t$, and weights in decoder layers. Embedding weights and the LM head usually stay in FP16 format and are not quantized. For decoder layers, only linear and layer norm layers contribute to memory. The memory requirement for quantized decoder layers with precision bit is calculated as $(4 \times h_1^2 + 2 \times h_1 \times h_2) \times \frac{4 \times bit}{32} + 6 \times h_1$ or $4 \times h_1$. For *KV cache*, we reserve it with a size equal to the sum of the maximum generated token length (a.k.a, context length) s and token generation number $n = t_{max}$. The memory size (in bytes) required by the KV cache for batched requests is estimated as $2 \times v(s+n)h_1 \times \frac{4 \times bit_{kv}}{32}$, where bit_{kv} is the bitwidth for representing elements in the KV cache. For *activation*, we assess the peak activation by considering the worst case scenario in embedding and decoder layers during prefill and decode phases.

Latency Cost Model. Computation intensity varies substantially across the prefill and decode phases. For instance, on an NVIDIA V100 GPU, the arithmetic intensity values (48, 43) during the decode phase for OPT-175B and 30B models with a batch size of 32 and a prompt length of 512 are notably lower compared to those (9553, 6354) in the prefill phase. This significant disparity in arithmetic intensity demands separate cost modeling for different phases.

Therefore, we model the execution time of the prefill phase as a function of FLOPs, based on v, s, vs and vs^2. The decode phase is dominated by memory access; we hence use the total number of bytes accessed (a.k.a, MOPs), to model decoding time, based on parameters $v, v(t+s)$ and $(t+s)$. We profile the execution time of each phase on one decoder layer under different precisions with common prompt lengths and batch sizes. We then use interpolation among the sample points to obtain a linear regression model for the execution time of one decoder layer in each phase.

B. Indicator of Model Perturbation by Quantization

We build performance indicators for low-precision weight-only kernels. INT8 kernel in this paper incurs little performance degradation [26], [35], we take the same indicator format with weight-only kernels for simplicity. State-of-the-art weight-only quantization of LLMs focuses on linear operators and [10], [11], [36] typically target the following objective:

$$\mathbf{Q}^* = \arg\min_{Q} \mathcal{L}(\tilde{\mathbf{W}}), \qquad \mathcal{L}(\tilde{\mathbf{W}}) = \|\mathbf{W}\mathbf{X} - \tilde{\mathbf{W}}\mathbf{X}\|_2^2 \quad (1)$$

Here $\mathcal{L}$ is the loss function, typically the minimum square error (MSE). $\mathbf{W}$ denotes the set of original FP16 weights of a decoder layer, and $\tilde{\mathbf{W}}$ is the set of quantized weights by quantization method Q, i.e., $\tilde{\mathbf{W}} = Q(\mathbf{W})$. $\mathbf{X}$ is the input feature, which refers to the layer input that corresponds to a small set of data points running through the network [11]. The goal is to identify the quantization method Q^* which minimizes the loss. Previous research [37] has used the eigenvalues of the Hessian matrix $\mathbf{H}$ of $\mathcal{L}$ with respect to $\mathbf{W}$ to measure a layer's sensitivity (error term) to quantization, as $\omega = \lambda \|Q(\mathbf{W}) - \mathbf{W}\|_2^2$, where λ is the top eigenvalue of Hessian $\mathbf{H}$. It requires computation of Hessian and quantization error ($\|Q(\mathbf{W}) - \mathbf{W}\|_2^2$) with respect to different precisions, incurring large computation overhead.

We adopt a different approach to describe a layer's sensitivity upon quantization. One key observation is that the quantization error originates from the $Round$ function. For a vector $\mathbf{x}$, $Round$ rounds each of its elements x to $\lfloor x \rfloor$ or $\lceil x \rceil$. We consider the round variance of quantization for two widely applied rounding methods, i.e., deterministic and stochastic [38], and derive an upper bound of the output variance introduced by quantization.

Theorem 1: The variance of a linear operator's output after weight-only quantization using stochastic or deterministic rounding is:

$$Var[\bar{\mathbf{W}}\mathbf{X}] = \begin{cases} Var[\mathbf{W}\mathbf{X}] + D_{\mathbf{W}} S_{\mathbf{W}}^2 \frac{1}{4} Var[\mathbf{X}], & \text{Deterministic} \\ Var[\mathbf{W}\mathbf{X}] + D_{\mathbf{W}} S_{\mathbf{W}}^2 \frac{1}{6} (\mathbb{E}[\mathbf{X}]^2 + Var[\mathbf{X}]), & \text{Stochastic} \end{cases} \quad (2)$$

where $D_{\mathbf{W}}$ is the dimension of model weights $\mathbf{W}$ and $S_{\mathbf{W}}$ is the scaling factor.

The theorem shows that the variance introduced by quantization in each linear operator is proportional to the dimension and scaling factor of the model weights. The scaling factor $S_{\mathbf{W}}$ is typically defined as $S_{\mathbf{W}} = \frac{\mathbf{W}max - \mathbf{W}min}{2^b - 1}$ (for asymmetric quantization), or $S_{\mathbf{W}} = \frac{max(abs(\mathbf{W}max), abs(\mathbf{W}min))}{2^{(b-1)} - 1}$ (symmetric quantization), where $\mathbf{W}max$ and $\mathbf{W}min$ are the largest and smallest weight values in $\mathbf{W}$. Given $\mathbf{W}$, the scaling factor is a function of quantization bitwidth b, denoted as $S_{\mathbf{W}}(b)$.

Proposition 1 (Variance Indicator): We measure the quantization sensitivity of a decoder layer i using the estimated quantization variance of the layer's output, i.e.,

$$\omega_{i,b} = \sum_{o}^{O_i} D_{\mathbf{W}_o} (S_{\mathbf{W}_o}(b_i))^2 G(\mathbf{X}_o) \quad (3)$$

where O_i is all linear operator within a layer, W_o represents the weight of linear operator o, X_o is the input feature, and $G(\mathbf{X})$ equals $\frac{1}{4} Var[\mathbf{X}]$ for deterministic or $\frac{1}{6}(\mathbb{E}[\mathbf{X}]^2 + Var[\mathbf{X}])$ for stochastic, respectively.

The variance indicator ω models the extra variance of output of a layer due to weight quantization. We use this indicator to rank the model performance impact of different quantization precisions for different layers. Operations in $G(\mathbf{X})$, i.e., mean and variance, are elementwise, with greatly reduced computation complexity as compared to Hessian calculation ($\mathcal{O}(D_{\mathbf{W}_i} D_{\mathbf{X}_i})$ vs. $\mathcal{O}(D_{\mathbf{W}_i} D_{\mathbf{X}_i}^2)$).

C. Optimizer

We propose an iterative algorithm that jointly optimizes three key parameters: quantization bitwidths, micro-batch sizes, and layer-to-device allocation, balancing the trade-off between inference latency and output quality degradation in LLM serving. Our approach systematically explores device topologies and micro-batch configurations across prefill and decode phases, then formulates and solves an integer linear programming (ILP) problem to derive layer partitions and bitwidth assignments under compute, memory, and quality constraints. For large-scale deployments or scenarios where ILP becomes computationally prohibitive, we introduce a lightweight heuristic that retains near-optimal solutions while reducing solve time by orders of magnitude.

Bitwidth Assignment and Layer Partition. We define a binary variable $z_{i,j,b}$ to indicate whether layer i is assigned to device j with quantization bitwidth b (1 for yes, 0 for no). The symbols B, η, and ξ represent the global batch size (or maximum concurrent requests [5]), the micro-batch size in the prefill phase, and the micro-batch size in the decode phase, respectively. Let L denote the number of layers in the LLM, and n the number of generated tokens. We assume input sequences within a batch are padded and dynamically chunked [21] into prompts of uniform length s, partitioned into κ chunks. The system comprises N devices indexed as $j \in \{1, 2, \ldots, N\}$, where M_j denotes the memory capacity of device j. BITs represents the set of available quantization bitwidths, e.g., BITs $= \{3, 4, 8, 16\}$.

$$\begin{aligned} \min_{\mathbb{Z}} \quad & (\lceil \frac{B}{\eta} - 1 \rceil T_{max}^{pre} + \lceil \frac{B}{\xi} - 1 \rceil (n-1) T_{max}^{dec} + T_{pre} + T_{dec}) \\ & + \theta \sum_{j=1}^{N} \sum_{i=1}^{L} \sum_{b \in BITs} z_{i,j,b} \omega_{i,b} \end{aligned} \quad (4)$$

The first parenthesized term in the objective (4) represents the end-to-end serving latency for a batch's token generation. In a pipeline-parallel serving system, the latency of serving a batch is the execution time of all pipeline stages plus $\mu - 1$ times the time taken by the slowest stage, where μ is the number of micro-batches [39]. In our LLM serving system, the end-to-end inference latency consists of the execution time of prefill and decode phases, corresponding to micro-batch numbers $\mu_{pre} = \lceil \frac{B}{\eta} \rceil$ and $\mu_{dec} = \lceil \frac{B}{\xi} \rceil$ for the two phases, respectively. Given n tokens to generate, the end-to-end latency is the sum of the prefill time of the first token and the decode time of the remaining $n-1$ tokens. This latency model can be readily adapted to handle *variable-output-length* scenarios by estimating token generation based on workload distribution. The second term in the objective function corresponds to the overall degradation in model quality, which we measure using a variance indicator.

$T_{max}^{pre}, T_{max}^{dec}, T_{pre}$ and T_{dec} are contingent upon $\mathbb{Z}$ (the vector of all decision variables $z_{i,j,b}$) as in constraints (5)-(8) , where $T_{\cdot,j}$ is execution time on device j, $T_{\cdot,j}^{max}$ is the maximum execution cost across pipeline stages. $l_{i,j,b}^{s,0}$ represents the average prefill computation time per-batch under prefill micro-batch size η, and $l_{i,j,b}^{s \times \kappa, \frac{n}{2}}$ is the average decode computation time per-batch under decode micro-batch size ξ, where i, j, b refers to the layer index, device index, and bitwidth, s is the prompt length. We halve the token number ($\frac{n}{2}$) for time estimation since decode cost increases linearly with each additional token in the past sequence for the next token. Costs are obtained from the latency cost models in Sec. IV-A. Communication in our system is asynchronous, as specified in constraint (7), P_{pre} and P_{dec} denote the transmission data size in the prefill and decode phases and f_j is the communication bandwidth between device j and its successor.

Constraints (9) - (11) ensure that only one bitwidth is assigned to a given layer and each layer can only be placed on a single device. Constraints (12)-(13) guarantee that memory consumption on each device j does not exceed its available memory capacity M_j (which is typically the GPU memory minus those consumed by cuda context), where $M_{i,b}^{s \times \kappa + n}$ denotes memory reservation according to the maximum sequence length, using our memory cost model. Constraint (13) of the first device in the given device ordering accommodates the memory requirement, M_{emb}, of embeddings for LLM pre or postprocessing as well. Constraints (15)-(16) ensure a continuous layer partition solution, as adjacent layer can be only placed on same or neighboring stage, where $u_{i,j}$ indicates whether layer i is placed on device j. We solve the ILP using

$$T^{pre}_{max} \geq T_{pre,j} = \sum_{i=1}^{L} \sum_{b \in BITs} z_{i,j,b} l^{s,0}_{i,j,b} \kappa, \quad \forall j = 1, ..., N \tag{5}$$

$$T^{dec}_{max} \geq T_{dec,j} = \sum_{i=1}^{L} \sum_{b \in BITs} z_{i,j,b} l^{s \times \kappa, \frac{n}{2}}_{i,j,b}, \quad \forall j = 1, ..., N \tag{6}$$

$$T^{pre}_{max} \geq \frac{P_{pre}}{f_j}, \quad T^{dec}_{max} \geq \frac{P_{dec}}{f_j}, \quad \forall j = 1, ..., N \tag{7}$$

$$T_{pre} = \sum_{j}^{N} T_{pre,j}, \quad T_{dec} = \sum_{j}^{N} T_{dec,j} \tag{8}$$

$$\sum_{j=1}^{N} \sum_{b \in BITs} z_{i,j,b} = 1, \quad \forall i = 1, ..., L \tag{9}$$

$$\sum_{j=1}^{N} z_{i,j,b} = y_{i,b}, \quad \forall i = 1, ..., L, b \in BITs \tag{10}$$

$$\sum_{b \in BITs} z_{i,j,b} = u_{i,j}, \quad \forall i = 1, ..., L, j = 1, ..., N, \tag{11}$$

$$\sum_{i=1}^{L} \sum_{b \in BITs} z_{i,j,b} M^{s \times \kappa + n}_{i,b} \leq M_j, \quad \forall j = 2, ..., N \tag{12}$$

$$\sum_{i=1}^{L} \sum_{b \in BITs} z_{i,1,b} M^{s \times \kappa + n}_{i,b} + M_{emb} \leq M_1 \tag{13}$$

$$y_{i,b}, u_{i,j}, z_{i,j,b} \in 0, 1, \quad \forall i = 1, ..., L, j = 1, ..., N, b \in BITs \tag{14}$$

$$u_{0,0} = 1, u_{L,N} = 1, \tag{15}$$

$$u_{i,j} + u_{i-1,k} \leq 1, \forall i = 2, ..., L, j = 1, ..., N-1, k = j, ..., N-1 \tag{16}$$

an off-the-shelf solver GUROBI [40].

Device Topology and Micro-batch Enumeration. We brute-force enumerate device topology orderings and micro-batch size combinations. Each device topology ordering represents a sequential arrangement of pipeline stages, where one stage maps to one physical device. All candidate orderings are generated by permuting available devices. For tensor parallelism (TP) [8], we restrict TP to intra-node configurations, enumerating valid 2D device meshes (e.g., 2×8, 4×4) [39] that respect node boundaries while generating permutations. The micro-batch size set $\mathcal{S}$ contains integers μ where $1 \leq \mu \leq B$. For each enumerated combination, we solve the corresponding ILP formulation to determine optimal quantization bitwidth assignments and layer partitioning.

Complexity of Algorithm. The solution space of the ILP problem in Equation (4) has size $\binom{L}{N} \cdot |Bits|^L$, where $\binom{L}{N}$ represents the number of possible layer partitions and $|Bits|$ denotes available bitwidth choices per layer. The algorithm requires at most $|\mathcal{G}| \cdot |\mathcal{S}|$ enumerations (line 1), resulting in a total search space complexity of $|\mathcal{G}| \cdot |\mathcal{S}| \cdot \binom{L}{N} \cdot |Bits|^L$. While this combinatorial explosion raises scalability concerns, we can efficiently address it with layer grouping and microbatch size pruning. While the solving time remains acceptable for static configurations (i.e., one-time cost per-model-per-cluster), the algorithm's computational cost grows exponentially with problem scale, creating a practical bottleneck. To address this, we propose a heuristic approach that efficiently generates high-quality solutions.

Heuristic: Bitwidth Transfer. Layer execution performance differs between GPUs while memory usage remains fixed. This allows precision conversion and layer partition adjustments between stages using transformation rules $\mathcal{C} \triangleq (b_{st}, b_{pi}, nums)$, which coordinate bitwidth conversion and layer repartitioning across pipeline stages. For example, $(4, 8, 2)$ replaces one 8-bit pioneer layer with two 4-bit straggler layers, improving precision or reducing layers to accelerate the slowest stage. Building on this framework, we develop a *bitwidth transfer* heuristic for solving the ILP problem (4). First, we remove the latency objective and solve a simplified ILP (*adabits*, compared in Sec. VI-H). We generate all valid transformations, identify the slowest stage (straggler), and iteratively apply adjustments to improve the objective value. This heuristic is effective in most cases.

TABLE III: Cluster Configurations

Cluster	Devices	Cluster	Devices
1	1xV100-32G	2	2xV100-32G + 1xA100-40G
3	1xV100-32G + 1xA100-40G	4	3xV100-32G + 1xA100-40G
5	3xT4-16G + 1xV100-32G	6	3xP100-12G + 1xV100-32G
7	4xT4-16G + 2xV100-32G	8	4xT4-16G
9	4xV100-32G	10	4xA100-40G

V. Implementation

SplitQuant integrates with the vLLM (v0.8.5.dev) [5] as a 964 LoC plugin, enabling phase-aware model partitioning and mixed-precision inference across modern quantization schemes—GPTQ [11], AWQ [10], and SmoothQuant [13]—alongside their optimized implementations (GPTQModel-Marlin [41], llmcompressor [42]). By building atop vLLM, our design fully leverages existing inference optimizations such as ChunkedPrefill [21], FlashAttention [43], and PagedAttention.

To bridge compatibility gaps in legacy low-calibre hardware (e.g., NVIDIA P100/T4 GPUs) and resolve edge cases in heterogeneous configurations that trigger vLLM backend failures, we implement a PyTorch [44]-native custom backend (4,655 LoC) that bypasses these limitations. For resource optimization, the Assigner component (1,355 LoC) integrates GUROBI's [40] ILP solver to automate allocation decisions.

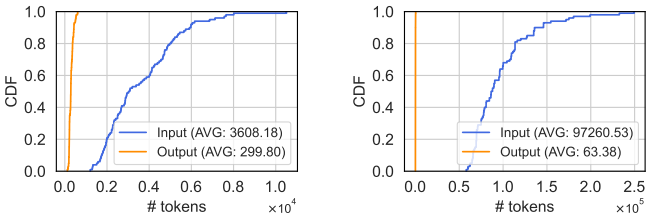
VI. Evaluation

A. Experimental Setup

Models & Precision. We evaluate four major open-source model families: Qwen2.5 [45], Llama-3 [23], and OPT [3], spanning model sizes from 7B to 70B parameters to reflect diverse serving scenarios. Specifically, we test Qwen2.5-7B-Instruct, Qwen2.5-14B-Instruct, Qwen2.5-32B-Instruct, OPT-30B, OPT-66B, and Llama3.3-70B-Instruct. We also test BLOOM [2] to show the fidelity of the cost model. We evaluate candidate precisions: $BITs = \{3, 4, 8, 16\}$, the 3-bit configuration is only configured by our custom quantization backend due to incompatibilities in the standard engine.

Baselines. We mainly compare with two baseline policies: (1) *Uniform*: A default configuration using uniform quantization with evenly partitioned model layers. (2) *Het*: Enumerate the parallelism schemes and applies uniform quantization but performs workload-aware balancing across pipeline stages for layer partitioning [12], [46]. For (1) and (2), we keep lowering the precision from the maximum (i.e., FP16) until the model can fit into the devices or no feasible solutions are available.

Workload. We focus on throughput-oriented offline workloads processed by dedicated inference servers. We choose two workloads: (1) *Summarization*, for which we sample prompts



(a) (b)

Fig. 7: The input and output length distribution of (a) CNN Dailymail Summarization and (b) LooGLE long context understanding.

from the CNN Dailymail dataset [22], [47] as inputs. (2) *Long Context Understanding*, where prompts are sampled from the LooGLE dataset [24]. As shown in Fig. 7, LooGLE has much longer input prompts (Avg. 97k) and shorter outputs (Avg. 63). For inference engine hyperparameter setup, we use 2048-token chunked prefill (Sarathi-Serve [21]), 256 max concurrent requests, and model-configured max sequence length. We then synthesize batches of size 256, filtering them based on the model's `max_position_embeddings` to ensure compatibility with the model's maximum context length. For the customized backend, considering the capacity, we synthesize a smaller workload following the setup in the DeepSpeed paper [19], with a batch size of 32, prompt size 512.

Metrics. We evaluate LLM serving performance by (1) output token throughput (tkn/s) and (2) model quality, averaging perplexity (PPL) on WikiText2 [28], Penn Treebank (PTB) [29], and C4 [30]. The weight calibration data consists of 128 randomly selected 2048 token segments from the C4 dataset [30].

Clusters. Devices/nodes are in our production cluster. We construct a number of heterogeneous clusters for model serving (clusters 2-7 in Table III), with a mix of common types of GPUs. GPUs of the same type are located on the same node, intra-connected with NV-LINK; Clusters 1,8,9,10 are on a single node, and others consist of two nodes. Nodes in Clusters 6,8 with 100Gbps Ethernet, others with 800Gbps Ethernet; Each node is equipped with two CPUs, P100 nodes with Intel Xeon CPU E5-2630 v4 2.2GHz, 64G RAM, V100 with Intel Xeon Gold 6230 2.1GHz, 128G RAM and 450G RAM, T4 with Intel Xeon Platinum 8260 CPU, 108G RAM, A100-40G with AMD EPYC 7H12 64-Core, 256G RAM. OS: Ubuntu 20.04.6 LTS. We also show SplitQuant's performance on several homogeneous clusters (clusters 8-10 in Table III).

Experiment Settings Whether to use the heuristic method and θ is hand-tuned by selecting values from the set $\{1, 10, 50, 100\}$ to trade off the solving time, solution quality target and the inference throughput. The problems have an average solving time of 18.38s (max: 115.981s) using GUROBI and the heuristic method.

B. Fidelity of Cost Models

We evaluate our memory cost model on BLOOM of sizes 560m and 1b7, and OPT of 13b, 30b, and 66b, with prompt length uniformly sampled between 128 and 512, the batch size chosen among 2, 4, and 8, generated token length sampled between 100 and 200, and randomly generated precision setting from the available bitwidth set. We consider the memory consumption of model weights and KV caching here and compare the predicted memory usage with those collected from real systems. We also create 50 unseen workloads with different precisions, batch sizes (3,5 or 7), prompt lengths, and past sequence lengths (384 or 768) for each device, evaluate our latency cost model on them. Fig. 8 shows that the error of the memory cost model is almost negligible, and the average error of the latency cost model is less than 6%.

Model	Real Mem (MB)	Est. Mem (MB)
BLOOM-560m	1821.14	1820.35
BLOOM-1b7	3430.20	3427.19
OPT-13b	24128.04	24121.43
OPT-30b	49380.95	49367.96
OPT-66b	73199.83	73166.66

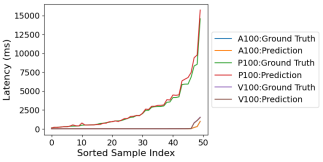
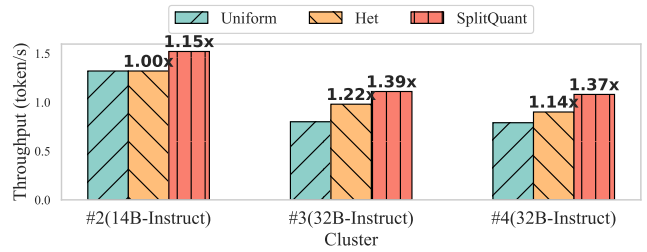


Fig. 8: Comparison of memory and latency reported by the cost models and obtained in real systems.

(a)



(b)

Fig. 9: End-to-end throughput comparison for heterogeneous devices on vLLM backend. (a) CNN Daily Mail Summarization and (b) LooGLE long context understanding. The speedup is derived by comparing with the Uniform baseline.

We observed that, during the prefill phase, the cost of observations typically increases linearly with the workload. However, it is noteworthy that in the decode phase, a notable difference in latency occurs only when a substantial change in context length (50-100) is present.

C. Serving in Heterogeneous Clusters

To isolate the impact of efficiency improvements from potential quality trade-offs, we constrain our solution to maintain at least the same model quality as the Uniform baseline. This configuration excludes interference from the quality scalar θ, which otherwise creates a trade-off between inference speed and model accuracy (see Sec . VI-G). Our results thus reflect pure efficiency gains.

Fig. 9 demonstrates that SplitQuant achieves an average 37% throughput improvement over the vLLM backend base-

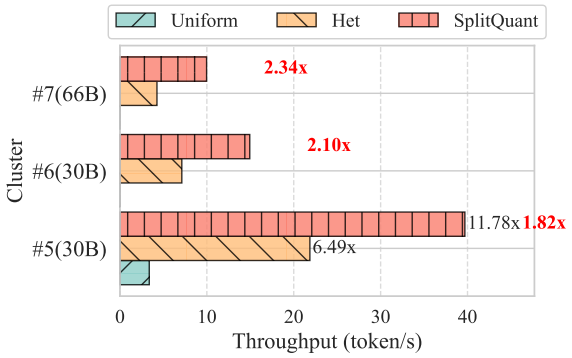


Fig. 10: End-to-end throughput comparison for heterogeneous devices on customized backend. The red speedup is derived by comparing with the Het baseline. 0 indicates OOM.

line. Notably, while the CNN DailyMail task exhibits significantly higher token generation throughput than LooGLE due to larger output sizes (299 vs. 63 tokens), both workloads show comparable gains under SplitQuant, highlighting its consistent performance across varying output lengths. The 14B-Instruct model results reveal that heterogeneous partitioning alone proves ineffective unless combined with adaptive mixed-precision optimizations.

Fig. 10 evaluates SplitQuant in a more severe heterogeneous cluster environment with legacy low-calibre GPUs. The Uniform baseline encounters OOM or fails to accommodate the model in most configurations due to device capability mismatches. Even when using basic heterogeneous partitioning (Het baseline), SplitQuant achieves an average 108% throughput improvement by exploiting phase-aware optimizations. These results suggest that increased hardware heterogeneity amplifies SplitQuant's optimization opportunities.

D. Serving in Homogeneous Clusters

On homogeneous clusters, Table IV demonstrates that SplitQuant continues to achieve throughput gains over baseline approaches, though these improvements are more modest compared to heterogeneous deployments. Clusters 9 and 10 exemplify scenarios where multiple valid GPU topology configurations exist. Specifically, Cluster 9 achieves optimal performance with a TP4 (Tensor Parallelism 4) configuration, while Cluster 10 performs best under a TP2+PP2 (Tensor Parallelism 2 combined with Pipeline Parallelism 2) arrangement. This divergence underscores the importance of dynamic device topology reordering, as discussed in Sec. IV-C—to automatically identify and select the most efficient hardware mapping for a given workload.

E. Effectiveness of Variance Indicator

To further validate the effectiveness of our variance indicator, we compare it with random assignment, where $\omega_{i,b}$ is assigned a value sampled from a uniform distribution. In the random indicator, we force higher bitwidth indicator values to be kept smaller than lower bitwidth indicator values within a layer. We also compare our indicator with Hessian-based as discussed in Section IV-B. We replace the indicator used in SplitQuant and adjust θ in (4) to ensure that different indicators lead to similar inference latency, eliminating the influence of value range of the indicator. In Table V, we observe that SplitQuant achieve better perplexity than FP16 on cluster 7. On cluster 8, Hessian-based and our indicators yield the same perplexity, outperforming the pure random indicator, while our indicator is much faster.

TABLE IV: Performance Comparison on CNN dataset in Homogeneous Clusters. The best throughput is marked in bold.

Cluster	Model	Scheme	Configuration	Throughput	Speedup
Cluster 1	7B-Instruct	Uniform	–	612.00	1.00×
		Het	–	N/A	N/A
		SplitQuant	–	**638.91**	**1.04**×
Cluster 9	70B-Instruct	Uniform	PP4	53.12	0.44×
		Uniform	TP2+PP2	83.12	0.69×
		Uniform	TP4	120.12	1.00×
		Het	TP4	120.12	1.00×
		SplitQuant	**Optimal**	**130.83**	**1.09**×
Cluster 10	70B-Instruct	Uniform	PP4	OOM	N/A
		Uniform	TP2+PP2	250.19	0.99×
		Uniform	TP4	211.68	0.84×
		Het	TP2+PP2	251.44	1.00×
		SplitQuant	**Optimal**	**291.25**	**1.16**×

TABLE V: Effectiveness of SplitQuant's variance indicator. PPL is compared with Random, while speedup is compared with Hessian.

Model	Cluster	Method	PPL	Overhead (s)
OPT-66b	7	Random	10.33	0
		Hessian	10.33	25625.44
		SplitQuant	**10.31(-0.02)**	**434.78(58.15**×**)**
OPT-30b	8	Random	11.04	0
		Hessian	10.75	15670.87
		SplitQuant	**10.75(-0.29)**	**215.60(72.69**×**)**

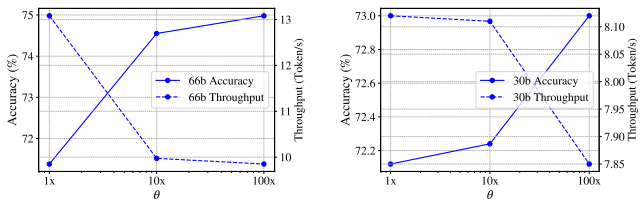
TABLE VI: Effectiveness of Grouping and Heuristic approaches under time limit. The best results are marked in bold.

Model	Cluster	Method	Throughput (token/s)	Overhead (s)
OPT-30b	5	Group=2	**39.70**	**1.07**
		Group=1	**39.70(+0)**	3.29
		Heuristic	35.17	5.36
OPT-30b	6	Group=2	14.72	12.29
		Group=1	13.93(-0.79)	204.59
		Heuristic	**14.94(+0.22)**	**1.99**
OPT-66b	9	Group=2	16.64	59.27
		Group=1	17.57(+0.93)	127.28
		Heuristic	**17.97(+1.33)**	**2.11**

F. Approaches Expediting Optimizer Algorithm

In SplitQuant, we provide two approaches, layer grouping and our *bitwidth transfer* heuristic, to reduce and the complexity of the optimizer's bitwidth selection, model partition, and placement. We evaluate the inference throughput and the time required to derive the solution when applying three strategies (*group = 2*, *group = 1*, and *heuristic*), on clusters 5, 6 and 9. *group = 2* means group 2 decoder layers together for decision. We set a **60-second** time limit for **each run** of the ILP solver. In our optimizer (See Sec. IV-C), the solver will be invoked several times.

Group = 1 covers the entire solution space and typically produces better results compared to *group = 2* (on clusters 9),



(a) (b)

Fig. 11: Sensitivity experiments on θ. (a) Cluster 7 OPT-66B. (b) Cluster 8 OPT-30B.

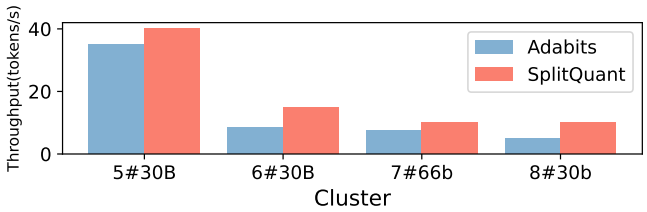


Fig. 12: Comparison with pure adaptive quantization.

but it introduces a larger overhead, as shown in Table VI. On cluster 6, *group = 1* cannot find a good solution within the time limit. On cluster 5, *group = 1* and *group = 2* produce the same solution. Performance of the heuristic largely depends on the starting point produced by *adabits* (start point of optimization #3 in Sec. IV-C). It leads to the best throughput with the smallest overhead in clusters 6 and 9. We highlight that the utilization of heuristics significantly enhances the **scalability** of SplitQuant in diverse GPU combinations.

G. Parameter Sensitivity

We next investigate the impact of user quality scalar θ in (4). We denote the value of θ we used in experiment in Sec. VI-D as $10\times$, scale it by 0.1 and 10 to obtain θ values of $1\times$, $100\times$. We evaluate model quality and serving throughput of SplitQuant under each θ value. Fig. 11 shows that a larger θ generally results in lower inference throughput and higher model accuracy, as less weight is placed on inference latency and more on model quality in our ILP optimization.

H. Ablation on Pure Adaptive Quantization

To verify the significance of concurrently considering adaptive bitwidth, layer partitioning, and micro-batch sizing, we further compare SplitQuant with *adabits* used in the heuristic method. We evaluate the performance of *adabits* with the same model setup on clusters 5,6,7, and 8. In Fig. 12 SplitQuant outperforms *adabits* in all selected cases, showing that **joint optimization** of partition and mixed-precision in SplitQuant offers optimal performance.

VII. Related Work

CPU-Assisted Inference. Prior work has explored leveraging auxiliary CPU resources co-located with accelerators to improve inference performance. For offline scenarios, systems such as FlexGEN [17], PowerInfer [48], and HeteGen [49] offload portions of compute or storage workloads (e.g., KV caches, activations) from GPUs to CPUs. Advanced approaches like Devotail [50] further investigate CPU-assisted speculative decoding. In contrast, SplitQuant optimizes *GPU-only* offline inference and does not rely on auxiliary CPU resources, making our work orthogonal to these efforts.

Heterogeneous GPU Serving. Several systems address LLM serving in heterogeneous GPU environments. For example, Helix [51] uses a max-flow problem for optimized scheduling, while HexGen employ asymmetric parallelism and graph partitioning. HexGen-2 [52] extends scenario to disaggregated environments via graph partitioning and flow optimization. However, these works do not systematically explore co-design with quantization techniques, a key focus of the optimization strategy of SplitQuant.

Quantization Methods for Large Language Models. Recent advancements in LLM quantization now explore precisions lower than 8-bit. Methods like Atom [53] focus on very low-precision configurations such as W4A4. Others, including QoQ [35] and QQQ [54], have improved W4A8 performance with hardware-aware designs. Despite these advances, deploying these methods on heterogeneous GPUs remains inefficient. We view them as candidate quantization schemes and can be effectively integrated into our framework.

VIII. Conclusion

We propose SplitQuant, an efficient system for LLM serving atop heterogeneous clusters. We derive efficient cost models to accurately predict memory occupation and execution latency of mixed-precision LLM serving. We introduce adaptive mixed-precision into the search space of pipeline serving and propose an efficient indicator to guide bitwidth selection in the search process. We jointly consider serving latency in different token generation phases based on various precision settings, micro-batch sizes, and layer partitions, and derive efficient optimized solutions. Our extensive experiments validate the performance of SplitQuant on a variety of cluster setups, which surpasses state-of-the-art approaches of serving LLM on heterogeneous clusters.

Acknowledgment

This work was supported in part by grants from Hong Kong RGC under the contracts 17204423, 17205824, C7004-22G (CRF), C5032-23G (CRF), and T43-513/23-N (TRS).

References

[1] OpenAI, "Chatgpt: Optimizing language models for dialogue," 2023.
[2] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, *et al.*, "Bloom: A 176b-parameter open-access multilingual language model," *ArXiv*, vol. abs/2211.05100, 2022.
[3] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," *ArXiv*, vol. abs/2205.01068, 2022.
[4] 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," *ArXiv*, vol. abs/2302.13971, 2023.
[5] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[6] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "SGLang: Efficient execution of structured language model programs," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[7] N. Corporation, "TensorRT-LLM Backend for Triton Inference Server." https://github.com/triton-inference-server/tensorrtllm_backend, 2023.

[8] 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.

[9] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[10] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," *ArXiv*, vol. abs/2306.00978, 2023.

[11] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *ArXiv*, vol. abs/2210.17323, 2022.

[12] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. Walters, "Pipeline parallelism for inference on heterogeneous edge computing," *ArXiv*, vol. abs/2110.14895, 2021.

[13] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International Conference on Machine Learning*, 2023.

[14] G. Hutt, V. Viswanathan, and A. Nadolski, "Deliver high performance ml inference with aws inferentia," 2019.

[15] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[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] 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*, 2023.

[18] RyokoAI, "Sharegpt52k," 2021.

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

[20] 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.

[21] 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.

[22] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Vancouver, Canada), pp. 1073–1083, Association for Computational Linguistics, July 2017.

[23] A. Grattafiori, A. Dubey, A. Jauhri, and et al., "The llama 3 herd of models," 2024.

[24] J. Li, M. Wang, Z. Zheng, and M. Zhang, "Loogle: Can long-context language models understand long contexts?," *arXiv preprint arXiv:2311.04939*, 2023.

[25] Z. Yao, R. Y. Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He, "Zeroquant: Efficient and affordable post-training quantization for large-scale transformers," in *Advances in Neural Information Processing Systems*, 2022.

[26] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm. int8 (): 8-bit matrix multiplication for transformers at scale," *arXiv preprint arXiv:2208.07339*, 2022.

[27] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker, "Perplexity—a measure of the difficulty of speech recognition tasks," *The Journal of the Acoustical Society of America*, vol. 62, no. S1, pp. S63–S63, 1977.

[28] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," 2016.

[29] M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of English: The Penn Treebank," *Computational Linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[30] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *arXiv e-prints*, 2019.

[31] D. Paperno, G. Kruszewski, A. Lazaridou, N. Q. Pham, R. Bernardi, S. Pezzelle, M. Baroni, G. Boleda, and R. Fernández, "The LAMBADA dataset: Word prediction requiring a broad discourse context," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Aug. 2016.

[32] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord, "Think you have solved question answering? try arc, the ai2 reasoning challenge," *arXiv:1803.05457v1*, 2018.

[33] Y. Bisk, R. Zellers, R. L. Bras, J. Gao, and Y. Choi, "Piqa: Reasoning about physical commonsense in natural language," in *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[34] E. Frantar, R. L. Castro, J. Chen, T. Hoefler, and D. Alistarh, "Marlin: Mixed-precision auto-regressive parallel inference on large language models," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 239–251, 2025.

[35] Y. Lin, H. Tang, S. Yang, Z. Zhang, G. Xiao, C. Gan, and S. Han, "Qserve: W4a8kv4 quantization and system co-design for efficient llm serving," 2024.

[36] T. Dettmers, R. Svirschevski, V. Egiazarian, D. Kuznedelev, E. Frantar, S. Ashkboos, A. Borzunov, T. Hoefler, and D. Alistarh, "Spqr: A sparse-quantized representation for near-lossless llm weight compression," *ArXiv*, vol. abs/2306.03078, 2023.

[37] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Hawq: Hessian aware quantization of neural networks with mixed-precision," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[38] B. Wan, J. Zhao, and C. Wu, "Adaptive message quantization and parallelization for distributed full-graph gnn training," *ArXiv*, vol. abs/2306.01381, 2023.

[39] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, *et al.*, "Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[40] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2023.

[41] ModelCloud.ai and qubitium@modelcloud.ai, "Gptqmodel." https://github.com/modelcloud/gptqmodel, 2024. Contact: qubitium@modelcloud.ai.

[42] vLLM Team, "Llm compressor: Efficient compression for large language models." https://github.com/vllm-project/llm-compressor, 2025.

[43] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," 2022.

[44] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Neural Information Processing Systems*, 2019.

[45] A. Yang *et al.*, "Qwen2.5 technical report," 2025.

[46] Y. Jiang, R. Yan, X. Yao, Y. Zhou, B. Chen, and B. Yuan, "Hexgen: Generative inference of large language model over heterogeneous environment," 2024.

[47] K. M. Hermann, T. Kociský, E. Grefenstette, L. Espeholt, W. Kay, M. Suleyman, and P. Blunsom, "Teaching machines to read and comprehend," in *NIPS*, pp. 1693–1701, 2015.

[48] Y. Song, Z. Mi, H. Xie, and H. Chen, "Powerinfer: Fast large language model serving with a consumer-grade gpu," 2024.

[49] X. Zhao, B. Jia, H. Zhou, Z. Liu, S. Cheng, and Y. You, "Hetegen: Heterogeneous parallel inference for large language models on resource-constrained devices," 2024.

[50] L. Zhang, Z. Zhang, B. Xu, S. Mei, and D. Li, "Dovetail: A cpu/gpu heterogeneous speculative decoding for llm inference," 2024.

[51] Y. Mei, Y. Zhuang, X. Miao, J. Yang, Z. Jia, and R. Vinayak, "Helix: Serving large language models over heterogeneous gpus and network via max-flow," 2025.

[52] Y. Jiang, R. Yan, and B. Yuan, "Hexgen-2: Disaggregated generative inference of llms in heterogeneous environment," 2025.

[53] Y. Zhao, C.-Y. Lin, K. Zhu, Z. Ye, L. Chen, S. Zheng, L. Ceze, A. Krishnamurthy, T. Chen, and B. Kasikci, "Atom: Low-bit quantization for efficient and accurate llm serving," in *Proceedings of Machine Learning and Systems*, 2024.

[54] Y. Zhang, P. Zhang, M. Huang, J. Xiang, Y. Wang, C. Wang, Y. Zhang, L. Yu, C. Liu, and W. Lin, "Qqq: Quality quattuor-bit quantization for large language models," 2024.