Evaluating Quantization Methods for LLM Inference on Low-VRAM Consumer GPUs

Erik Wasmosy - CSC 420

December 8, 2025

1 Introduction

Generative AI models like Large Language Models (LLMs) [1] require a huge number of parameters to perform well on different tasks and to gain a generalized understanding of human knowledge [2]. However, making them practically deployable is significantly constrained by the memory these models need. For instance, running inference on a 7-billion (7B) parameter model requires around 28GB of VRAM [3], and a 7B model is considered relatively small nowadays. Newer models have hundreds of billions of parameters, requiring multiple GPU models for both training and inference. Even for a smaller 7B model, only a limited number of GPU models can hold that many parameters, which are typically enterprise-grade cards (like the A100 or H100) or the latest top-tier graphics cards (like an RTX 4080 with 16GB of VRAM or better).

This places these powerful models outside the reach of the vast majority of GPUs found in personal desktops and laptops, as they simply contain less VRAM. To get past this "VRAM Barrier" a common solution is to quantize the model's parameters by reducing their numerical precision. Weights can be reduced to formats like half-precision (FP16), 8-bit floating point (FP8), or even 4-bit formats (FP4). Since precision is being cut, the main trade-off of quantization is a potential loss of accuracy in the model's outputs [4].

To combat this trade-off, for example 4-bit formats have various versions designed to maximize accuracy, such as P4 (E2M1), MXFP4, and the newest NVFP4 introduced in NVIDIA's Blackwell architecture [5]. This format achieves high accuracy by using an innovative dual-scale system (FP32 and FP8); its local FP8 scale allows for fractional adjustments, minimizing quantization error far better than other FP4 formats.

Consequently, many companies and startups rely on calling third-party REST APIs to get generative AI content. This is because maintaining the required amount of server compute carries a significant financial risk. However, a high volume of user requests can lead to huge costs in API calls. To reduce this cost without building a server warehouse, an alternative is to deploy models on locally-hosted hardware. This could include gaming PCs, laptops, or Mini-PCs with capable GPUs, such as those with AMD iGPUs or Apple silicon, which feature unified memory that provides ample capacity and bandwidth for loading models [6][7]. The GGUF file format can help with this, as it is designed for storing and running LLMs on consumer-grade hardware, is efficient for the inference process, and allows splitting the model's layers between the GPU and CPU [8].

This project shows experiments running LLMs with modern weight quantization techniques on a laptop with an RTX 5070 Max-Q GPU[9], which has 8GB of VRAM. In this study, we tried two of the most popular engines to run LLMs: vLLM, a state-of-the-art Python library for running LLMs at a large scale (focusing on the highest throughput and simultaneous requests possible)[10], and Llama.cpp, which focuses on running efficient transformer models on an edge device [11].

With our limited resources, we found that vLLM is hard to get a model up and running or to switch models without significant tweaks. However, with the models that we successfully loaded into the GPU, we got good performance in terms of tokens per second, even with multiple simultaneous requests. On the other hand, Llama.cpp (thanks to the GGUF file format and different quantization methods) was a lot easier for running a broad variety of language models, making it easier to set up an end-to-end process, which we also show in the results section.

2 Background

VRAM limitations when running LLMs are well known in the field, specifically within the Transformer architecture on which they are based[12]. These limitations are particularly critical during the autoregressive decoding step and attention

algorithms, specifically involving the KV cache. This dynamic data, the keys and values for previous tokens, grows with every new token generated. Traditional systems waste significant memory due to fragmentation because they must pre-allocate large, contiguous blocks for this cache. This contiguous layout also makes it difficult to efficiently share memory, such as sharing the cache for common prompt prefixes across different requests[13].

A primary method for reducing the memory usage of LLMs at inference time is quantization, which is divided into two main types: Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ)[14]. We are focusing on PTQ, which relies on taking a model's parameters after it has been trained and reducing their precision, often with the trade-off of some accuracy loss[4].

The vLLM inference engine contains state-of-the-art techniques for minimizing the problems of memory usage described above with its own attention algorithms like PagedAttention [10], including different Quantization methods[15]: BitBlas [16], GPTQModel (General-purpose Post-Training Quantization) [17], AWQ (Activation-aware Weight Quantization)[18], and moreover, they also recently analyzed NVIDIA's new Blackwell architecture, which introduced NVFP4 for 4-bit floating point operations, resulting in a 4x throughput speed [19].

The primary goal of these quantization methods (such as AWQ and GPTQ) and the vLLM inference engine is to allow a model to be loaded entirely into one or more GPUs to achieve maximum throughput. While some libraries, such as HuggingFace's transformers, offer options like device_map="auto" to split a model between the CPU and GPU [20], this is not the primary design for high-performance engines. On the contrary, vLLM's PagedAttention mechanism [13] is fundamentally an advanced VRAM management technique, reinforcing its GPU focus.

On the other hand, there is Llama.cpp, an engine more focused on running LLMs in edge AI devices [21], where usually these devices do not have a GPU, only CPU. If it has one, it is a small one that is known as a Neural Processing Unit (NPU), which can be found on smartphones or modern regular laptops, but usually they have their specific scenarios to have parallelism operations with low power consumption [22], and it can be helpful too for computers with a consumer GPU, like a gaming GPU where the VRAM is limited. The developer of Llama.cpp also created the underlying ggml library and the GGUF file format; these are what enable Llama.cpp to thrive on the edge hardware.

Instead of requiring a large amount of VRAM, Llama.cpp leverages ggml to perform inference efficiently on CPUs, including ARM. A key part of this is the use of GGUF format files, which are optimized for faster loading of models and which also support quantized tensors. While ggml's primary strength is on CPU performance, it also has the ability to split the workload (e.g., model's layers) to GPUs to get extra performance, making it ideal for scenarios with limited VRAM.

GGUF files have their own nomenclature for quantization versions of models, following this pattern: Q[Number]_[Type]_[Size]. Q[Number] is the number of bits used per weight (e.g., Q4 = 4-bit, Q5 = 5-bit, and Q8 = 8-bit). For the [Type], it could be _0 or _1, but these are legacy types; they are straightforward quantization methods and lower quality than _K types (which are the modern recommended balance between speed and accuracy) and i-Quants types, which focus on getting better quality at low bit sizes based on which weights are more important, but can be slower than K-quants. And lastly, the _[Size] indicates the block size and precision mix; this could be _M for medium (which offers a good balance), _S, _L, or even _XSS [8].

On this project, we got a lot better out-of-the-box experience using Llama.cpp with GGUF files than with vLLM and using other quantization methods, primarily because Llama.cpp automatically offloads the model's weights and computation between the CPU and GPU, while with vLLM, it is hard to make it run if the model weights and KV cache do not fit in your VRAM without any tweaks.

3 Method

In this work, first we built the vLLM project from source code to have available the latest updates and be able to use the latest and most optimized backends for NVIDIA GPUs, and also try models with support of NVFP4 and see if our GPU would be able to handle that. Since we had the newest Blackwell architecture GPU, we checked NVIDIA's CUDA GPU Compute Capability Website to see if the laptop GPU would support it [23]. The Blackwell family has a Compute Capability of 12.0, which supposedly supports nvfp4. However, the site doesn't specify if this support (e.g., for the RTX 5070 Max-Q) also applies to the laptop versions. We wrote a small CUDA program and confirmed that our GPU does indeed report Compute Capability 12.0.

Then, to have a better development experience,

we changed the focus to use Llama.cpp since it made it easier to run the models we want without spending too much time tweaking and trying many different arguments, as was the case with vLLM for this project.

With Llama.cpp, we served two LLM models at the same time to receive requests for two different kinds of tasks: one was an Optical Character Recognition (OCR) transformer-based model and the other was a Text Generation LLM. With that, we wanted to experiment with a scenario where modern AI applications could use this setup to reduce the time for manual forms completion. For this, we wrote a small project in Golang to call into this server and complete the workflow. This scenario is where a tenant needs to upload a proof of payment screenshot of different bank apps for his rent payment to his landlord, without making the tenant fill out input fields again where he has to enter the transaction date, the total amount, and the currency. Thus, a call is first made to the OCR server, and then we pass the extracted text to a text generation model to output a JSON with these three fields. This is a small example, but it could be useful in larger scenarios where you have a contract and it has a lot of information and you need to extract the relevant information.

In the results section, we show the performance of a quick test of the throughput using an LLM on vLLM and a quick test on NVFP4. We then present a broader test on Llama.cpp, including the accuracy for the same models with different quantization versions of the GGUF files, to see how far we can make them smaller without losing too much accuracy.

3.1 Dataset

For testing the small AI application feature we mentioned before, we used our own dataset that we collected from previous users of their proof of payment for one month of rent. After cleaning some duplicate values or data that was used for developing the software, we ended up with 40 examples from real users. We labeled these png, jpeg, or pdf files with three main features: transaction date, total amount, and currency, and used them to get the accuracy. We also included another feature just to check if there are reasons for failing, which is image quality (e.g., a screenshot is a 'perfect' picture, while a blurry picture or a picture of a receipt could be considered medium or low). So if a test failed, we could justify that it was for the picture quality or for other reasons, such as multiple dates showing in the picture and the Generative model getting confused. We could then improve the prompt to improve its ability to determine which one it has to select as the transaction date. Table 1 shows a quick summary of the data set, the fields, and their range of values.

Table 1: Dataset Summary and Value Ranges. The bold headers (**Amount**, **Currency**, **Date**) are used to measure model accuracy.

File Name	**Date**	**Amount**	**Currency**	Quality
.pdf, .png, .jpeg, .jpg	Oct 01 – Nov 25 '25	Monthly Rent Cost	USD, PYG	High, Med, Low

4 Results

In this section, first we describe what we tried with vLLM and a quick experiment with NVFP4 running with vLLM. Unfortunately, tweaking and getting the right models to run something in the GPU was a hard experience and time consuming, so these subsections are preliminary results. The models we wanted to run on both vLLM and Llama.cpp didn't work on vLLM to try to compare the throughput. To meet our deadlines, we moved completely after the first presentation to Llama.cpp to be able to continue with the project and get some results to benchmark the small feature for a modern AI application that we described in the previous section.

After the preliminary results, we show results running on Llama.cpp with AI models stored in GGUF files. We list the models we tried, parallelized requests, time benchmarks, and accuracy. For accuracy, we got good results. However, the parallelized OCR process was complicated. It was noticed that the total processing time scaled linearly with the number of requests; for example, two requests took twice as long as one, and four requests took four times as long.

4.1 Preliminary results: vLLM and NVFP4

4.1.1 vLLM

We began by serving a small model, facebook/opt-125m [24], which ran successfully using the vLLM Python library – part of the quickstart guide [25].

Then, we attempted to serve a larger model with 1.5 billion parameters, the Qwen2.5-1.5B-Instruct [26] model, on our 8GB VRAM machine. This initial run failed with an Out-of-Memory (OOM) error during server initialization. The logs suggested vLLM's warmup phase was the cause, as it attempted to allocate memory for 256 single-token dummy requests simultaneously. The default value for running vLLM as a server

is max_num_seqs=256. This significant upfront memory attempt exceeded the available VRAM.

This behavior, where initialization fails but actual serving is efficient, suggests the warmup allocation test bypasses the normal operational scheduler. During normal operation, the scheduler manages request batching incrementally, constrained by both concurrent requests and total tokens per batch, which is far more VRAM-efficient. As a temporary fix, we reduced max_num_seqs from 256 to 100, which successfully lowered the warmup memory requirement, allowing the server to initialize.

To validate the server's stability under load, a Go script was developed to send 50 concurrent requests using diverse programming task prompts (to prevent prefix caching). The vLLM scheduler managed these requests by batching them efficiently, demonstrating stable performance under high concurrency, which suggests the underlying issue is specific to the warmup phase's large, non-scheduled memory allocation, not the scheduler's ability to handle real requests.

Table 2: vLLM Server Performance (Qwen2.5-1.5B on RTX 5070 Max-Q 8GB VRAM) under Concurrent Load

Requests	Peak Prompt (tok/s)	Peak Gen. (tok/s)	Peak KV Cache (%)	Gen / Request
1	4.2	40	0.4	40
10	37	589	2.5	**59**
20	74	686	6.4	34
30	111	1058	9.1	35
50	185	1265	18.5	25
70	244	1796	22.9	26
80	**279**	1949	29.6	24
90	273	**2213**	**34.2**	25

4.1.2 First attempts on NVFP4

As we mentioned before, this project was tested on a Blackwell GPU in a laptop. Laptop versions of desktop GPUs are considered Max-Q versions [27], which are sold with the same name but are less powerful. So we weren't sure if this version had FP4 units.

After many attempts at running FP4 open-source models, and a lot of errors. Some were because the available models we found on the Nvidia profile were too big for our machine, and other FP4 models we found ran with FP4 Marlin, so we got this message: 'Your GPU does not have native support for FP4 computation', which is not exactly NVFP4. After checking the source code [28], we realized that they change the backend depending on whether libraries like FlashInfer, TensorRT-LLM, or CUTLASS are installed [29][30][31]. Fig. 1 shows the code section where the backend is selected. After installing these, it worked on our machine. Additionally, setting environment variables such as VLLM_NVFP4_GEMM_BACKEND=1 and ENABLE_NVFP4_SM120=1 made it work.

```python
class ModelOptNvFp4LinearMethod(LinearMethodBase):
    """Linear method for Model Optimizer NVFP4.
    Supports loading NVFP4 checkpoints with the following structure:

    input_scale: torch.float32, scalar ,
    weight: NVFP4(represented as byte) Shape: [1, X, y/2]
    weight_scale: FP8-E4M3, Shape: [X, Y], aka per block scale,
    weight_scale_2: torch.float32, scalar,
    Args: quant_config: The ModelOpt quantization config.
    """

    def __init__(self, quant_config: ModelOptNvFp4Config) -> None:
        self.quant_config = quant_config
        if envs.VLLM_USE_TRTLLM_FP4_GEMM:
            assert has_flashinfer(), "TRTLLM FP4 GEMM requires FlashInfer"
            self.backend = "flashinfer-trtllm"
        elif has_flashinfer():
            self.backend = "flashinfer-cutlass"
        elif cutlass_fp4_supported():
            self.backend = "cutlass"
        elif is_fp4_marlin_supported():
            self.backend = "marlin"
        else:
            raise ValueError("Current platform does not support NVFP4"
                             " quantization. Please use Blackwell and"
                             " above.")
```

Figure 1: Python implementation of `ModelOptNvFp4LinearMethod` showing dynamic selection of the backend to support FP4 operations such as TensorRT-LLM, FlashInfer, Cutlass, or Marlin, based on the current execution environment and hardware support.

Unfortunately, the model we tried was an 8 billion parameter model (nvidia/Qwen3-8B-NVFP4)[32], and it already took 6 GB for the weights, so there wasn't enough space for the KV cache. As you can see in the results in Table 3, we were not able to serve more than 10 requests at the same time. Also, the throughput is slower than the previous results because this is a larger model, and the accuracy might be better. In the future, we expect to find NVFP4 models that are smaller and take up less space, to see whether we can achieve better results, such as throughput, compared to the previously analyzed non-quantized model.

Table 3: Performance Results running the model nvidia/Qwen3-8B-NVFP4

Requested → Running	Generation Throughput (tok/s)	KV Cache (%)	peak gen / num of request
1 → 1	32.5	28.9	32.5
10 → 8	316	**99.5**	39.5
20 → 7	**382.3**	97.6	**54.6**

4.2 Llama.cpp, GGUF files, and AI application experiment

After we had tried a lot of times with vLLM, we decided to change the focus to Llama.cpp since it has a better focus on running LLMs on edge devices [11]. Here, we were able to move faster, we were able to try a lot of different models without any issues or tweaking, and get the things done.

To test the small flow described in section 3 (which is: extract the characters of an image with an OCR model and generate an output structure with a text generative LLM model), we first looked for the newest OCR models. We attempted to use Deepseek-OCR [33], but it exceeded our available VRAM, so we utilized LightOnOCR-1B-1025 instead, a vision–language model for OCR which was just released in October 2025, achieving state-of-the-art results by being fast and light [34]. We tested a version with imatrix quantization of the model created by the user 'noctrex' in huggingface [35]. And for the text generative model, we used qwen2.5-1.S-instruct-q4_k_m.gguf', since it performed well and it was a 4-bit model we hadn't tried any others by that time.

Model Type	**Models Tested**
OCR	LightOnOCR-1B-1025-i1-BF16 LightOnOCR-1B-1025-i1-F16 LightOnOCR-1B-1025-i1-Q6_K_M LightOnOCR-1B-1025-i1-Q4_K_M LightOnOCR-1B-1025-i1-Q3_K_M LightOnOCR-1B-1025-i1-Q2_K
Text Gen	Qwen2.5-Qwen2.5-1.5B-Instruct-Q4_K_M

Table 4: List of models tested organized by task type.

OCR	**TextGen**	**Date**	**Amount**	**Currency**
L-BF16	Q-Q4	81.67	**95.00**	97.50
L-F16	Q-Q4	**82.50**	93.33	95.83
L-Q6_K_M	Q-Q4	81.67	90.83	98.33
L-Q4_K_M	Q-Q4	**82.50**	90.83	97.50
L-Q3_K_M	Q-Q4	80.83	93.33	96.67
L-Q2_K	Q-Q4	78.33	92.50	**99.17**

Table 5: Accuracy. L: LightOnOCR, Q-Q4: Qwen2.5-1.5B-Q4_K_M

In table 4 we list the OCR and Text Generative models we tested, and in Table 5 we show the results of the end-to-end workflow and average after running three times, since LLMs are non-deterministic. This table lists the percentage of parts of the images we were able to identify correctly from our dataset. For detecting the currency, it did pretty well for all models. More than 90% of the dataset images are Gs/Guaranies currency and the rest are USD/dollar. Surprisingly, with the smallest OCR model, we got 99% of accuracy, but this does not mean that other OCR quantized versions did not get this result; it also depends on the text generative model and if it correctly grabbed the currency of what the OCR model extracted. For the amount field, accuracy was more than 90% for every OCR, and the rent for guaranies is around 7 numbers; here, with the largest model, we got the best performance. And finally, the date was where the models struggled the most. When we saw the models that failed on this field, we noticed it was from both models: from the OCR, it sometimes missed the number (instead of 17, the OCR output an 11), and from the Qwen model, it had more than one date to consider from the images and grabbed the current one, such as the distinction of 'transaction date' or 'processed date' data that some banks have in their app, which made the language model confused.

Finally, for timing, we tried to do parallel output with llama.cpp and vLLM, but when two requests or more arrived at the same time, the processing time was increasing linearly, so testing sequentially would have gotten the same wait time. This is a future work to fix, but our intuition is that the issue comes from the LightOnOCR and not from text generative models, since in vLLM and in Llama.cpp the text generative models were able to parallelize well and reduce time to serve multiple requests. With that said, in the Table 6 we show the times for a non-parallel request, one by one processing at a time.

It is also good to mention that, Llama.cpp allows you to split the model layers between CPU and GPU, but in these scenarios, we were able to load and run completely on the GPU both models at the same time. With this, we got a VRAM usage up to 5.6GB of 8GB available; the language model only occupied 1.2GB and the OCR 4GB or more in the VRAM.

We also tested running `llama.cpp` in parallel, processing 10 images at the same time. But, as shown in Table 7, this increased the time a lot for every request/user. The throughput got better a little better, from 0.48 to 0.77 req/s, but the average total time went from 2.1s to 12.6s. You can see the problem is the OCR time (1.8s to 11.5s), not the text gen (0.28s to 1.04s). We followed the `llama.cpp` documentation to do this correctly, but there are still some issues. We think this is more about the LightOnOCR model structure rather than the `llama.cpp` engine.

5 Conclusion and future work

We tested two modern plug-and-play LLM serving engines, vLLM and Llama.cpp, to see how easy it was to run Large Language Models on a low-VRAM GPU machine, and how good of an idea

OCR Model	Text Gen	Total Process Time (s)	OCR Time (s)	Text Generation Time (s)
LightOnOCR-BF16	Q4_K_M	Avg: 2.72 ± 2.94 Min: 1.55, Max: 24.91	Avg: 2.44 ± 2.89 Min: 1.30, Max: 24.25	Avg: 0.28 ± 0.05 Min: 0.23, Max: 0.67
LightOnOCR-F16	Q4_K_M	Avg: 2.73 ± 3.00 Min: 1.54, Max: 25.25	Avg: 2.46 ± 2.95 Min: 1.29, Max: 24.56	Avg: 0.28 ± 0.06 Min: 0.24, Max: 0.70
LightOnOCR-Q6_K_M	Q4_K_M	Avg: 2.00 ± 1.48 Min: 1.34, Max: 16.81	Avg: 1.73 ± 1.44 Min: 1.10, Max: 16.16	Avg: 0.27 ± 0.04 Min: 0.23, Max: 0.65
LightOnOCR-Q4_K_M	Q4_K_M	Avg: 2.10 ± 2.16 Min: 1.32, Max: 15.63	Avg: 1.82 ± 2.10 Min: 1.08, Max: 14.94	Avg: 0.28 ± 0.06 Min: 0.24, Max: 0.69
LightOnOCR-Q3_K_M	Q4_K_M	Avg: 1.81 ± 0.70 Min: 1.33, Max: 8.19	Avg: 1.54 ± 0.68 Min: 1.09, Max: 7.71	Avg: 0.27 ± 0.03 Min: 0.23, Max: 0.48
LightOnOCR-Q2_K	Q4_K_M	Avg: 2.39 ± 3.07 Min: 1.27, Max: 15.95	Avg: 2.10 ± 2.98 Min: 1.04, Max: 15.26	Avg: 0.29 ± 0.09 Min: 0.24, Max: 0.72

Table 6: Processing time statistics (average ± standard deviation, minimum, and maximum) for different quantization levels across 3 runs of 40 processed files.

Table 7: Sequential vs. Parallel (10 Requests) Performance Comparison for LightOnOCR-Q4_K_M and Qwen2.5-1.5B-Q4_K_M.

Metric	Sequential (1 Request)	Parallel (10 Requests)
Throughput	0.48 req/s	**0.77 req/s**
Total Process (s)		
Avg ± Std	2.10 ± 2.16	**12.61 ± 5.82**
Min, Max	1.32, 15.63	**4.19, 30.81**
OCR (s)		
Avg ± Std	1.82 ± 2.10	**11.58 ± 5.76**
Min, Max	1.08, 14.94	**3.49, 29.39**
Text Gen (s)		
Avg ± Std	0.28 ± 0.06	**1.04 ± 0.41**
Min, Max	0.24, 0.69	**0.51, 2.22**

it is to be useful as a server, avoiding being dependent on cloud services or paying for LLMs via curl API calls. We found that vLLM is more ready for big-scale production, while being hard to run a wider range of models if you have a limited device, with a lot of failures when running or tweaking to make it work. But with what we were able to test, we got good throughput for serving multiple requests. And we found that Llama.cpp could be more useful for running models while developing your application and also for serving AI applications for small features where the primary focus is to reduce time, for example, to extract text and use this to auto-complete large forms.

Also, we noticed that running Llama.cpp in our system, while parallel processing improved the overall system throughput (more requests per second), it significantly increased the latency for each individual user. This means that while the system handles more work, every user waits much longer than they would in a sequential, first-come-first-served queue. This trade-off remains a future work in progress, and further investigation is needed to see if other models, resources or configuration changes can solve this latency problem.

Additionally, other AI experiments or use cases need to be tested, such as AI agents, which could be more complicated in this project since AI agents need to have more 'intelligence,' meaning larger models and better throughput to generate a lot of tokens to 'think' and make a decision. However, on current applications, we noticed that they still take time to process, so the user might be waiting to continue working or just let the agent work and do other things in the meantime. So in those scenarios where background jobs are allowed in the user experience, it could be a good test to see if larger models will be effective on this kind of machine and leverage the CPU RAM if they are limited in VRAM.

References

[1] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2023.

[2] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[3] NVIDIA Corporation. GPU memory essentials for AI performance. https://developer.nvidia.com/blog/gpu-memory-essenti

als-for-ai-performance/, 2024. Accessed: 2025-10-30.

[4] Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. A comprehensive evaluation of quantization strategies for large language models, 2024.

[5] NVIDIA Corporation. Introducing NVFP4 for efficient and accurate low-precision inference. https://developer.nvidia.com/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/, 2024. Accessed: 2025-10-30.

[6] AMD Corporation. Unified memory: HIP 6.2.41133 documentation. https://rocm.docs.amd.com/projects/HIP/en/docs-6.2.0/how-to/unified_memory.html, 2024. Accessed: 2025-10-30.

[7] Apple Inc. Choosing a resource storage mode for Apple GPUs. https://developer.apple.com/documentation/metal/choosing-a-resource-storage-mode-for-apple-gpus, 2024. Accessed: 2025-10-30.

[8] Hugging Face. GGUF — huggingface.co. https://huggingface.co/docs/hub/gguf. [Accessed 27-11-2025].

[9] TechPowerUp. NVIDIA GeForce RTX 5070 Mobile Specs. https://www.techpowerup.com/gpu-specs/geforce-rtx-5070-mobile.c4237, 2025. [Accessed 07-12-2025].

[10] vLLM Team. vLLM documentation. https://docs.vllm.ai/en/latest/, 2024. Accessed: 2025-10-30.

[11] GitHub - ggml-org/llama.cpp: LLM inference in C/C++ — github.com. https://github.com/ggml-org/llama.cpp. [Accessed 28-11-2025].

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[13] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention, 2023.

[14] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. A survey on model compression for large language models. *Transactions of the Association for Computational Linguistics*, 12:1556–1577, 11 2024.

[15] vLLM Team. Quantization in vLLM. https://docs.vllm.ai/en/stable/features/quantization/, 2024. Accessed: 2025-11-21.

[16] Microsoft Corporation. BitBLAS: Mixed-precision matrix multiplication library for quantized LLM deployment. https://github.com/microsoft/BitBLAS, 2024. Accessed: 2025-11-21.

[17] ModelCloud. GPTQModel: LLM quantization toolkit with hardware acceleration. https://github.com/ModelCloud/GPTQModel, 2024. Accessed: 2025-11-21.

[18] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration, 2024.

[19] vLLM Team. SemiAnalysis InferenceMAX: vLLM and NVIDIA accelerate blackwell inference. https://blog.vllm.ai/2025/10/09/blackwell-inferencemax.html, 2025. Accessed: 2025-10-30.

[20] Hugging Face. Big Model Inference. https://huggingface.co/docs/accelerate/usage_guides/big_modeling. Accessed: 28-11-2025.

[21] IBM Corporation. What Is Edge AI? https://www.ibm.com/think/topics/edge-ai. Accessed: 29-11-2025.

[22] Josh Schneider and Ian Smalley. What is a Neural Processing Unit (NPU)? https://www.ibm.com/think/topics/neural-processing-unit. Accessed: 29-11-2025.

[23] NVIDIA Corporation. CUDA GPUs: Compute capability. https://developer.nvidia.com/cuda-gpus, 2024. Accessed: 2025-11-21.

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

[25] vLLM Team. Quickstart guide: vLLM. https://docs.vllm.ai/en/stable/getting_started/quickstart.html, 2024. Accessed: 2025-11-11.

[26] Qwen Team. Qwen2.5-1.5B-Instruct model. https://huggingface.co/Qwen/Qwen2.5-1.5B-Instruct, 2024. Accessed: 2025-11-21.

[27] NVIDIA Corporation. NVIDIA GeForce RTX 50 Series Laptops. https://www.nvidia.com/en-us/geforce/laptops/50-series/, 2025. [Accessed 07-12-2025].

[28] vLLM Project. vLLM: High-throughput and memory-efficient inference engine for LLMs. https://github.com/vllm-project/vllm, 2024. Accessed: 2025-11-21.

[29] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving, 2025.

[30] NVIDIA Corporation. TensorRT-LLM documentation. https://docs.nvidia.com/tensorrt-llm/index.html, 2024. Accessed: 2025-11-21.

[31] NVIDIA Corporation. CUTLASS: CUDA templates for linear algebra subroutines. https://docs.nvidia.com/cutlass/latest/, 2024. Accessed: 2025-11-21.

[32] NVIDIA Corporation. Qwen3-8B-NVFP4 model. https://huggingface.co/nvidia/Qwen3-8B-NVFP4, 2024. Accessed: 2025-11-21.

[33] Deepseek Team. deepseek-ai/DeepSeek-OCR · Hugging Face — huggingface.co. https://huggingface.co/deepseek-ai/DeepSeek-OCR. [Accessed 07-12-2025].

[34] Said Taghadouini, Baptiste Aubertin, and Adrien Cavaillès. LightOnOCR-1B: The Case for End-to-End and Efficient Domain-Specific Vision-Language Models for OCR. https://huggingface.co/blog/lightonai/lightonocr, 2025. Accessed: 01-12-2025.

[35] noctrex. noctrex/LightOnOCR-1B-1025-i1-GGUF. https://huggingface.co/noctrex/LightOnOCR-1B-1025-i1-GGUF. Accessed: 01-12-2025.