Implementing Generative AI Using Intel Xeon 6 CPUs

Planning / Implementation

Generative AI has emerged as a transformative force across industries, driving innovations in content creation, automation, and decision-making. From large language models (LLMs) powering chatbots to AI-generated media and drug discovery, the ability to generate human-like text, images, and solutions is reshaping how businesses operate. Traditionally, the field of Generative AI has relied heavily on GPUs due to their ability to perform massive parallel computations required for training and inferencing large-scale models. However, this dependence comes with significant challenges, including high costs, limited supply, energy consumption, and infrastructure complexity.

While frontier models such as GPT-4 and Gemini have demonstrated impressive capabilities as general-purpose AI systems, they come with substantial drawbacks. Their tremendous computational requirements necessitate the use of costly GPUs making them expensive to train and deploy, and their generalized nature can limit accuracy in specialized domains. Additionally, these large-scale models are often unable to integrate proprietary data effectively due to their closed-source nature, making them less adaptable for enterprise applications. In contrast, domain-specific models (DSMs) offer a more efficient alternative, focusing on specific industries or applications. DSMs also require fewer computational resources, making them more cost-effective while achieving higher accuracy in their specified domains. Their smaller size enables easier deployment on CPUs, and organizations can enhance their capabilities by incorporating proprietary data through retrieval-augmented generation (RAG) or post-training fine-tuning.

Intel's advancements in AI-optimized CPUs have further enabled efficient inferencing and, in some cases, even training on general-purpose processors. Intel Xeon 6 processors, equipped AI acceleration technologies (e.g., Intel AMX, AVX-512), large amounts of high-speed last-level cache, and many cores, are increasingly capable of handling Generative AI workloads with competitive performance and lower cost. By leveraging CPUs for AI deployment, organizations can take advantage of existing infrastructure, reduce capital expenditures, and simplify system integration while maintaining efficiency.

This paper presents a reference architecture for deploying Generative AI models on clusters of Intel CPU powered Lenovo servers, providing a cost-effective and scalable alternative to GPU-based systems. The following sections will guide readers through key aspects of building such a system, including model selection, fine-tuning, optimization for inference, and deployment strategies using vLLM and Docker.

Intel CPUs for Generative AI

The Intel Xeon 6 processors, integrated into Lenovo ThinkSystem SR650 V4, provide a powerful and scalable platform for generative AI workloads. With higher core density, double the memory bandwidth, and upgraded P-cores, Intel Xeon 6 processors ensure efficient execution of large-scale AI inference. Combined with the SR650 V4 which offers enterprise-grade reliability and optimized power efficiency, it is an ideal choice for AI deployments without GPU reliance.

An integral part of the Intel Xeon 6 processor, Intel Advanced Matrix Extensions (AMX) and AVX-512 significantly accelerate deep learning workloads by optimizing matrix multiplications and vectorized computations. These enhancements improve transformer-based model execution, enabling faster and more efficient AI inferencing on CPUs.

The OpenVINO Toolkit enables hardware-aware AI inference optimizations, including model compression, quantization (FP16, INT8, INT4), and operator fusion. These optimizations reduce memory footprint, lower latency, and maximize CPU performance, making OpenVINO a key enabler of cost-effective, scalable AI solutions.

System Setup

In this section we provide steps to set up the system environment which will be useful for the rest of the paper. This includes installing Docker, creating a Python virtual environment, and saving a Huggingface token.

Topics in this section:

Docker Install

The following steps describe how to install Docker on Ubuntu 24.10. For more details on Docker installations, go to https://docs.docker.com/engine/install/.

1. Set up Docker's apt directory

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/ap
t/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/d
ocker.asc] https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}")
stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

2. Install the Docker package

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx
-plugin docker-compose-plugin
```

3. Verify Docker runs correctly

```
sudo docker run hello-world
```

Python Virtual Environment

We recommend installing the uv package to create a virtual python environment to avoid dependency issues and quickly download needed python libraries. Steps are as follows:

1. Run the following in your Linux terminal:

```
curl -LsSf https://astral.sh/uv/install.sh | sh

uv venv openvino-env
source openvino-env/bin/activate
```

2. Activate Virtual environment and install requirements

```
source openvino_env/bin/activate
uv pip install vllm
uv pip install optimum-intel[openvino]
```

HuggingFace Setup

Steps to setup HuggingFace are as follows:

1. Install the huggingface Command Line Interface (CLI)

```
Uv pip install -q "huggingface_hub[cli]"
```

2. Login to your huggingface account using your token.

```
huggingface-cli login --token "token_here"
```

vLLM with OpenVINO Docker Image

The vLLM Docker image needs to be built from the source using the OpenVINO plugin.

1. Issue the following commands

```
git clone https://github.com/vllm-project/vllm-openvino.git
cd vllm-openvino
docker build . -t vllm-openvino-plugin-docker-img
```

Model selection

Several factors must be considered when selecting a model which we broadly categorize into two groups: Application and Compute criterion. Application criteria gauge how suitable a model is for our specific application, for instance performance on relevant datasets and multi-modal capabilities. Compute criteria depend on the compute power available as well as model size and architecture; this includes metrics such as throughput and latency.

We recommend looking through models available on huggingface where over a million pre-trained models are freely available: https://huggingface.co/. Additionally, models already optimized using OpenVINO can be found here: https://huggingface.co/OpenVINO

Topics in this section:

Application Considerations

First and foremost, a model must align with the intended application and its data characteristics. The following factors guide the evaluation process:

- **Task-Specific Benchmarking:** Selecting a model requires assessing its accuracy and efficiency on relevant datasets. For example:
    - **Question Answering:** Benchmarks such as MMLU (Massive Multitask Language Understanding) and AGIEval help to gauge model performance on textual reasoning and question-answering tasks.
    - **Code Generation & Reasoning:** HumanEval and MBPP (Mostly Basic Python Problems) measure a model's ability to generate and complete code.
    - **Translation:** FLORES-200 and WMT (Workshop on Machine Translation) are widely used benchmarks for evaluating translation quality across multiple languages, measuring accuracy with metrics like BLEU and ChrF.
- **Multimodal Requirements:** Some applications require models that can process and generate multiple data types (e.g., text, images, and audio). If the use case involves text-to-image generation (e.g., product design, creative content generation), a vision-language model like BLIP or OpenCLIP may be necessary. If only text-based tasks are required, a language model optimized for CPU inference, such as LLaMA-3 or Falcon, may be preferable.
- **Legal and Licensing Considerations:** Deploying generative AI models in production requires careful evaluation of licensing terms, intellectual property restrictions, and compliance with regulatory frameworks. Some models, such as LLaMA-3, may have light usage restrictions for commercial applications, while others, like Falcon or Mistral, offer permissive open-source licenses. Additionally, AI-generated content in regulated industries (e.g., healthcare, finance) may require provenance tracking and human oversight to mitigate risks.

In most cases, relevant benchmark performance, multi-modal capabilities, and licensing details are all provided with the model on huggingface or an associated publication.

Compute Considerations

When selecting a generative AI model for deployment on CPUs, key compute-related factors must be evaluated to ensure optimal performance. Unlike GPUs, CPUs excel in general-purpose compute and optimized inference workflows but require careful tuning to achieve low latency and high throughput. The following performance metrics are critical for evaluating model suitability:

- **Time-to-First-Token (TTFT):** Measures the delay between sending an inference request and receiving the first generated token. Lower TTFT is critical for interactive applications like chatbots and real-time summarization. CPU-based optimizations (provided through OpenVINO) and reduced model precision (e.g., INT8 quantization) can help minimize TTFT.
- **Tokens Per Second (TPS):** Represents the rate at which a model generates tokens once inference begins. Higher TPS is essential for throughput-bound applications such as document generation, machine translation, and bulk content processing. CPU-based optimizations and reduced model precision can again improve TPS.
- **Memory Footprint:** Larger models require more RAM, and while more abundantly available than on GPU deployments, large footprints can limit parallel processing capability on CPU-based deployments.
- **Batch Size and Concurrency:** CPUs can efficiently handle multiple concurrent inference requests if batch processing is optimized. Workloads with high request volumes benefit from advanced batching strategies such as continuous batching, where multiple input sequences are processed in a single inference pass.

Hands-on Testing

To ensure a selected model meets performance expectations for a given application, hands-on testing is essential. This section provides step-by-step guidance on setting up and conducting basic inference tests, measuring key performance metrics such as time-to-first-token (TTFT), tokens per second (TPS), and memory footprint on CPU-based deployments. In this demonstration, we will use an 8 billion parameter Llama-3.1 model.

By running the vLLM image we created earlier, we can host our Llama-3.1 model as well as many other freely available models in a containerized web server which will respond to OpenAI API requests on a specified port.

1. First, we start the container, allowing it to share its 8000 port with the host machine and mount the default directory where huggingface models are stored for easy access:

   ```
   docker run -it --rm -p 8000:8000 \
       -v ~/.cache/huggingface:/root/.cache/huggingface \
       -e HF_TOKEN= \
       -e CUDA_VISIBLE_DEVICES="" \
       vllm-openvino-plugin-docker-img
   ```

2. With the Docker container running, we may start the vLLM engine and begin serving the model. For this example we keep the command simple, simply listing the model we wish to host, the maximum model length, and specify the cpu as the target device. vLLM has an enormous number of features and a complete list of command line arguments is provided here: https://docs.vllm.ai/en/latest/serving/engine_args.html.

   ```
   vllm serve "meta-llama/Llama-3.1-8B-Instruct" \
       --max-model-len 32768 \
       --device cpu \
       --disable-log-requests
   ```

3. After a few moments the model should be running on the newly created server and be capable of receiving requests through a curl command. We can verify the web server is running and able to receive requests using the following command in a separate command line window. It should indicate the Llama-3.1-8B model is available.

```
curl http://localhost:8000/v1/models
```

4. Once we have verified the server is operational, we can make inferencing calls to the model using the OpenAI Completions API. We provide an example below but encourage the reader to experiment with their own prompts and observe both the output and the token generation speeds.

```
curl http://localhost:8000/v1/chat/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "meta-llama/Llama-3.1-8B-Instruct",
        "max_tokens": 100,
        "messages": [
            {"role": "system", "content": "You are a helpful assistant.
"},
            {"role": "user", "content": "Tell me about the benefits of
using Intel Processors."}
        ]
}'
```

5. Once we have adequately determined a selected model generates meets our application requirements, we can benchmark TTFT and TPS metrics to ensure it meets our compute requirements. The vLLM package contains several useful benchmarking scripts, including benchmark_serving.py. This script will make inference requests to the API using prompts from a sample dataset and time the response.

```
# Download vllm codebase
git clone https://github.com/vllm-project/vllm.git

# Download ShareGPT Dataset
curl -L https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_
unfiltered/resolve/main/ShareGPT_V3_unfiltered_cleaned_split.json -o Sh
areGPT_V3_unfiltered_cleaned_split.json
```

6. Now we can run the benchmarking script by specifying several command line arguments including: the model, port, dataset path, number of prompts, and request rate. The number of prompts and request rate determines how many API calls will be made and at what rate. An infinite rate means all API requests are made simultaneously.

```
python3 vllm/benchmarks/benchmark_serving.py --host localhost --port 80
00 --endpoint /v1/chat/completions --backend openai-chat --model meta-l
lama/Llama-3.1-8B-Instruct --dataset-path ShareGPT_V3_unfiltered_cleane
d_split.json --num-prompts 10 --request-rate inf

Example Output:
============ Serving Benchmark Result ============
Successful requests:                        10
Benchmark duration (s):                     59.23
Total input tokens:                         13693
Total generated tokens:                     2193
Request throughput (req/s):                 0.17
Output token throughput (tok/s):            37.02
Total Token throughput (tok/s):             60.14
---------------Time to First Token----------------
Mean TTFT (ms):                             1491.89
Median TTFT (ms):                           1491.60
P99 TTFT (ms):                              1496.59
-----Time per Output Token (excl. 1st token)------
Mean TPOT (ms):                             79.11
Median TPOT (ms):                           78.73
P99 TPOT (ms):                              81.89
---------------Inter-token Latency----------------
Mean ITL (ms):                              77.90
Median ITL (ms):                            76.86
P99 ITL (ms):                               95.70
```

Model Customization

After selecting a base model, LLMs can be further tailored to specific business needs. The ideal method of optimization varies based on the specific use case as well as the amount of time and resources a company can dedicate. The three primary methods of customization—Prompt Engineering, Retrieval-Augmented Generation (RAG), and Fine-Tuning—fall along a spectrum of increasing complexity and effectiveness.

Topics in this section:

Prompt Engineering

Prompt Engineering is the process of crafting input prompts to guide the model's responses without modifying its parameters. By carefully structuring prompts, users can achieve significant improvements in model performance with minimal effort. This technique is especially useful for applications where real-time adaptability is required or where model modifications are impractical or costly.

Here are a couple examples using Alpaca prompting:

Translation Prompt:

```
### Instruction: Translate the input from English to Spanish
### Input: {user input}
### Response:
```

Sentiment Analysis Prompt:

```
### Instruction: Analyze the sentiment of the following customer review and c
lassify it as Positive or Negative.
### Input: {user input}
### Response:
```

We can enhance the effectiveness of the prompt using a technique known as *few-shot learning*, where the model is provided with a few examples of the desired output format within the prompt. This helps the model generalize better to new queries without requiring retraining.

For example:

```
### Instruction: Analyze the sentiment of the following customer review and c
lassify it as Positive or Negative.

Example 1:
Review: "This laptop is amazing! Super fast, great battery life, and the scr
een is beautiful."
Sentiment: Positive

Example 2:
Review: "Terrible experience! It stopped working after just two days, and cus
tomer service was unhelpful."
Sentiment: Negative
Now, classify the sentiment of this review:

### Input: {user input}
### Response:
```

Retrieval-Augmented Generation (RAG)

RAG enhances generative models by integrating an external retrieval mechanism, such as a search engine or a vector store, allowing the system to fetch relevant document chunks or structured data before generating responses (see the figure below). This method is particularly effective when factual accuracy or domain-specific knowledge is required.

Instead of relying solely on the model's pretrained knowledge, RAG dynamically retrieves information from an external database or knowledge source. This allows for up-to-date and contextually relevant responses, addressing the model’s limitations in factual recall.

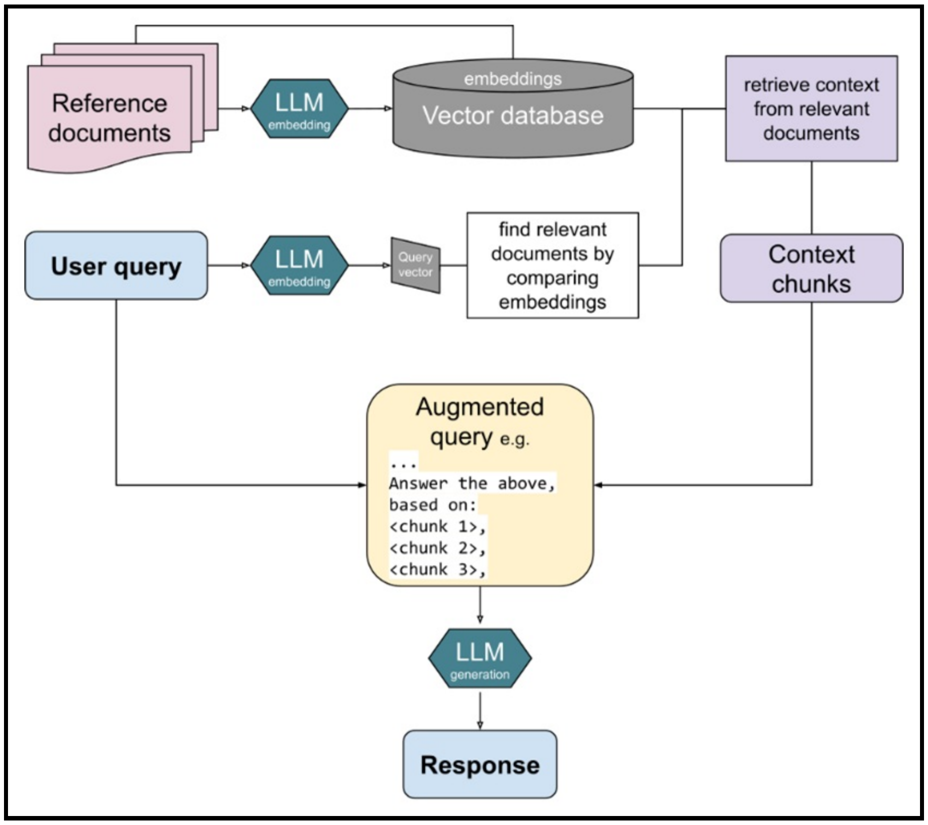

Figure 1. Retrieval-Augmented Generation

Fine-Tuning

Fine-tuning involves training an LLM on a domain-specific dataset to adapt its behavior, tone, and knowledge to a particular application. Unlike Prompt Engineering and RAG, which operate without modifying the model's internal weights, fine-tuning directly updates the parameters of the model, leading to deeper customization and improved performance in targeted tasks.

Fine-tuning is the most resource-intensive customization approach, but it is the best choice when:

- The model needs to align with a specific domain (e.g., legal, medical, or technical fields) that is not well-represented in its original training data.
- The model must exhibit a distinct brand voice, style, or company-specific terminology.
- High accuracy and consistency are required for tasks where prompt engineering and retrieval-based methods fall short.
- The model should reduce hallucinations by refining its knowledge within a limited scope.

For a more in-depth discussion on implementing fine-tuning, readers can refer to our previous paper on this topic:
https://lenovopress.lenovo.com/lp2179-fine-tuning-llms-using-intel-xeon-cpus

Choosing the Right Approach

When deciding on a customization strategy, planners should consider the amount of training data available as well as the cost to train and implement the method. It's generally best to start by trying the simplest/easiest solution, checking if it meets the application's needs, and increasing the complexity if needed. We list the available approaches in order of increasing complexity and direct the reader to Meta's recommendation flowchart shown in the figure below.

- **Prompt Engineering** - Quick and cost-effective; works well for simple adaptations.
- **RAG** - Useful when external knowledge is required without retraining the model.
- **Fine-Tuning** - Necessary for deep customization, improved accuracy, and specific formatting requirements.

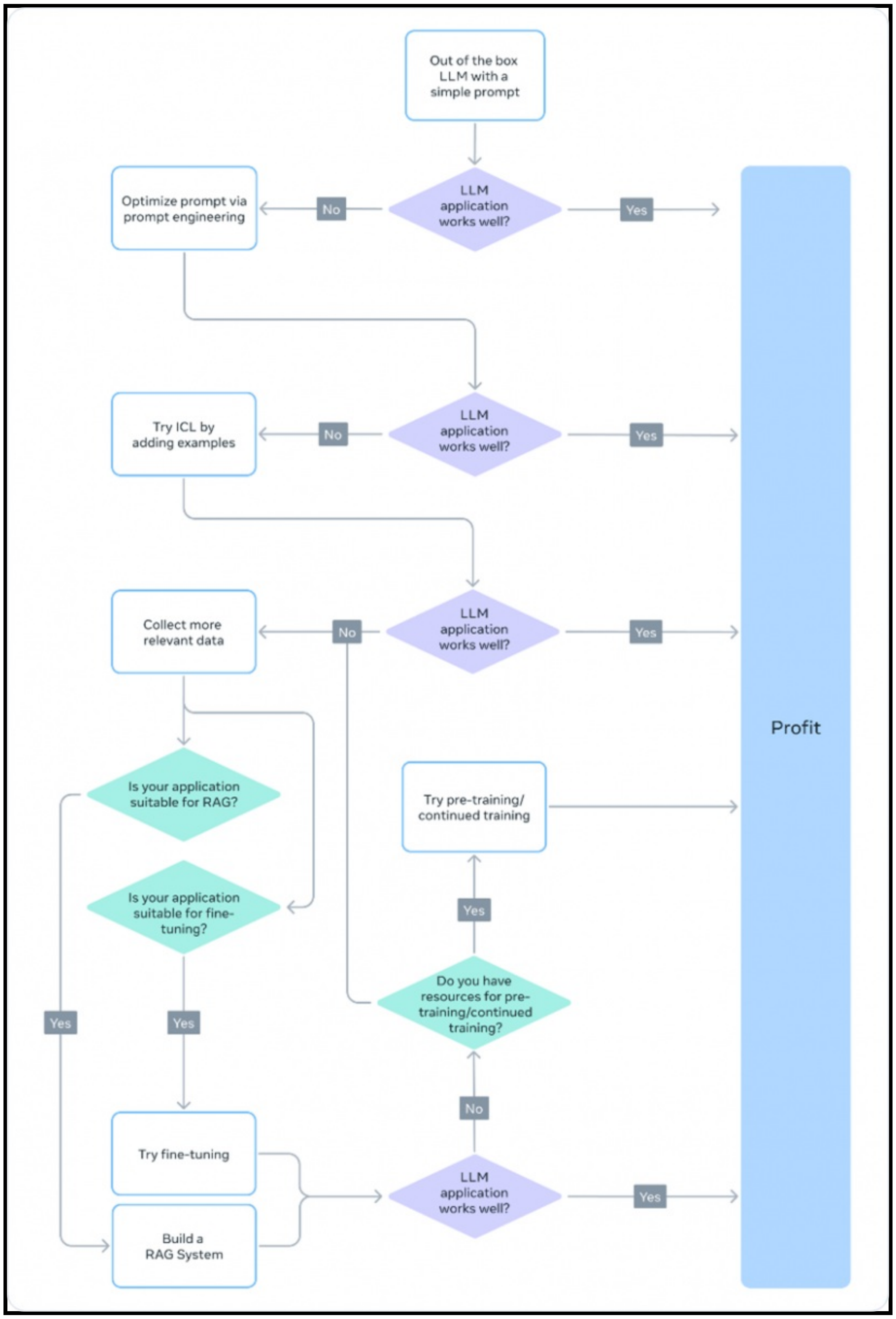
Out of the box LLM with a simple prompt
LLM application works well?
No
Yes
Optimize prompt via prompt engineering
LLM application works well?
No
Yes
Try ICL by adding examples
LLM application works well?
No
Yes
Collect more relevant data
Is your application suitable for RAG?
Is your application suitable for fine-tuning?
Yes
Yes
Try fine-tuning
Build a RAG System
LLM application works well?
No
Yes
Do you have resources for pre-training/continued training?
Yes
Try pre-training/ continued training
Profit

Figure 2. Choosing the Right Approach

Optimization for Inference

Once a base model has been selected and tailored, further optimizations can be applied to achieve lower latency, higher throughput, and efficient memory utilization onboard Intel CPUs. OpenVINO provides techniques to accelerate LLM inference on Intel CPUs. This section explores key optimization methods and best practices for deploying LLMs efficiently using OpenVINO.

Topics in this section:

- Weight Compression
- Implementing Weight Compression

Weight Compression

Optimizing the memory footprint of Large Language Models (LLMs) is essential for efficient deployment, particularly on resource-limited hardware. Weight compression reduces model size and computational resources used, while maintaining inference accuracy by focusing exclusively on weight optimization, as opposed to full quantization, which modifies both weights and activations.

For instance, compressing an 8-billion parameter Llama 3 model using 4-bit quantization can reduce its size from approximately 16.1 GB to 4.8 GB, significantly lowering memory requirements without extensive accuracy trade-offs.

Implementing Weight Compression

Compressing models is simple using the Optimum command line interface with OpenVINO support. After being weight-compressed, models can be deployed in the same manner as the original models. We can simply use the export openvino command and specify the model, the desired weight format, and the save location. The weight format includes most standard formats (fp32, fp16, int8, int4).

Deeper control is available with more options listed here:
https://huggingface.co/docs/optimum/en/intel/openvino/export

```
optimum-cli export openvino \
    --model meta-llama/Llama-3.1-8B-Instruct \
    --weight-format int8 \
    ~/.cache/huggingface/hub/meta-llama/Llama-3.1-8B-Instruct-OV-INT8
```

Model Deployment and Serving

Once we have selected and optimized our model, deployment in vLLM is as simple as deploying it in a Docker container in the same manner as we did before.

```
# Start the container
docker run -it --rm \
-p 8000:8000 \
-e CUDA_VISIBLE_DEVICES="" \
-v ~/.cache/huggingface:/root/.cache/huggingface \
vllm-openvino-plugin-docker-img

# Inside the container
vllm serve "/root/.cache/huggingface/hub/meta-llama/Llama-3.1-8B-Instruct-OV-
INT8" \
        --max-model-len 32768 \
        --device cpu
```

At this point we have successfully deployed a containerized LLM capable of responding to OpenAI Completion API requests. However the average user may find it cumbersome to make requests using curl commands, so we may use OpenWebUI to provide a more natural user interface. We can again containerize the application using Docker.

The important settings to note are the OPENAI_API_BASE_URL which should point to where we direct the OpenAI Completion API requests and the port we host the webpage on. In this instance we can access the OpenWebUI site at http://localhost:8080 but can be accessed remotely depending on your router configuration.

```
docker run -d -p 3000:8080 \
--network host \
-v open-webui:/app/backend/data \
--name open-webui \
--restart always \
--env=OPENAI_API_BASE_URL=http://localhost:8000/v1 \
--env=OPENAI_API_KEY=token-abc123 \
--env=ENABLE_OLLAMA_API=false \
ghcr.io/open-webui/open-webui:main
```

After creating an admin profile, you may now service new users in a clean UI similar to that of ChatGPT with a wide variety of features and extensions including web search and RAG search.

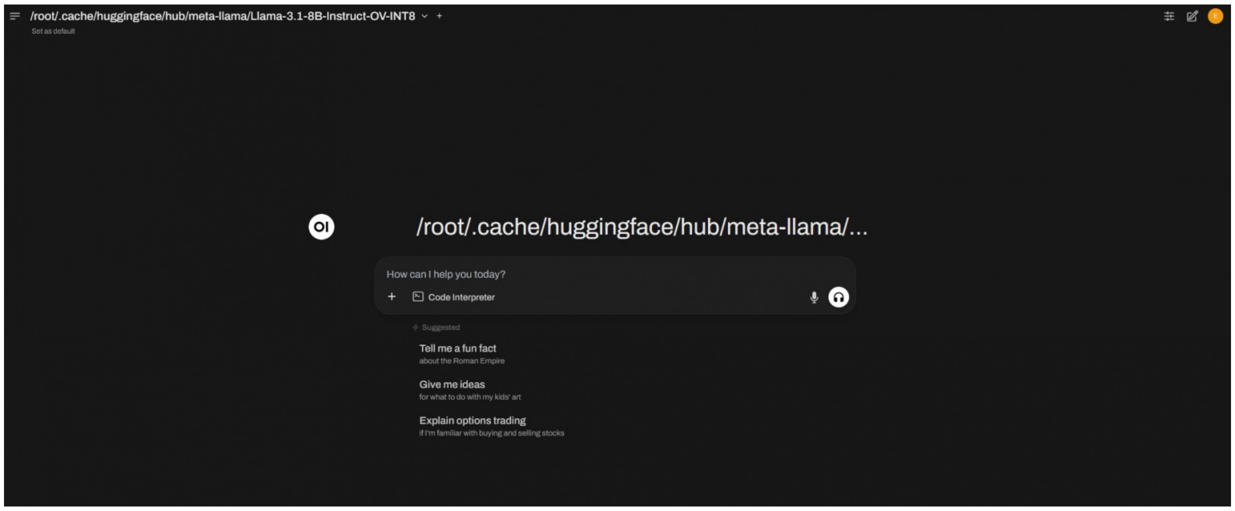
/root/.cache/huggingface/hub/meta-llama/Llama-3.1-8B-Instruct-OV-INT8
Set as default
/root/.cache/huggingface/hub/meta-llama/...
How can I help you today?
Code Interpreter
Suggested
Tell me a fun fact
about the Roman Empire
Give me ideas
for what to do with my kids' art
Explain options trading
if I'm familiar with buying and selling stocks

Figure 3. OpenWebUI Landing Page

Performance

Here we showcase the serving performance of the Intel Xeon 5 8592+ processor with and without OpenVINO acceleration as well as the estimated performance of the next generation Xeon 6700 with Performance-cores. We use vllm's benchmark_serving.py script mentioned earlier in the paper with Llama3.1-8B using the default parameters of 1000 requests and an infinite request rate.

Our results indicate OpenVINO more than doubles the throughput over the vanilla vllm-cpu docker container and we estimate a 68% improvement using the Intel Xeon 6700 over the previous generation.

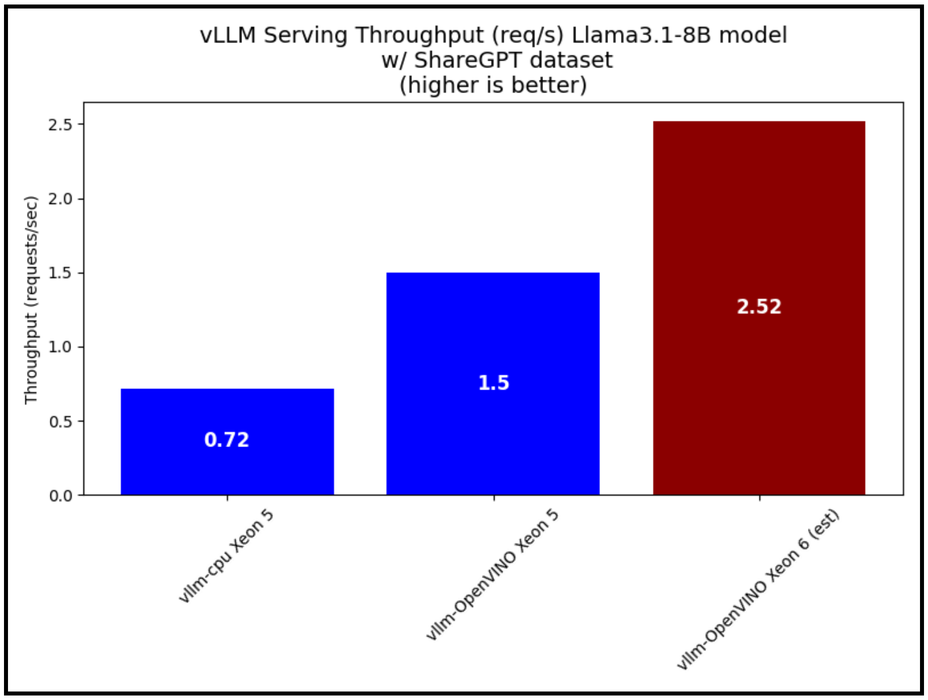

Figure 4. vLLM Serving Comparison (requests per second)

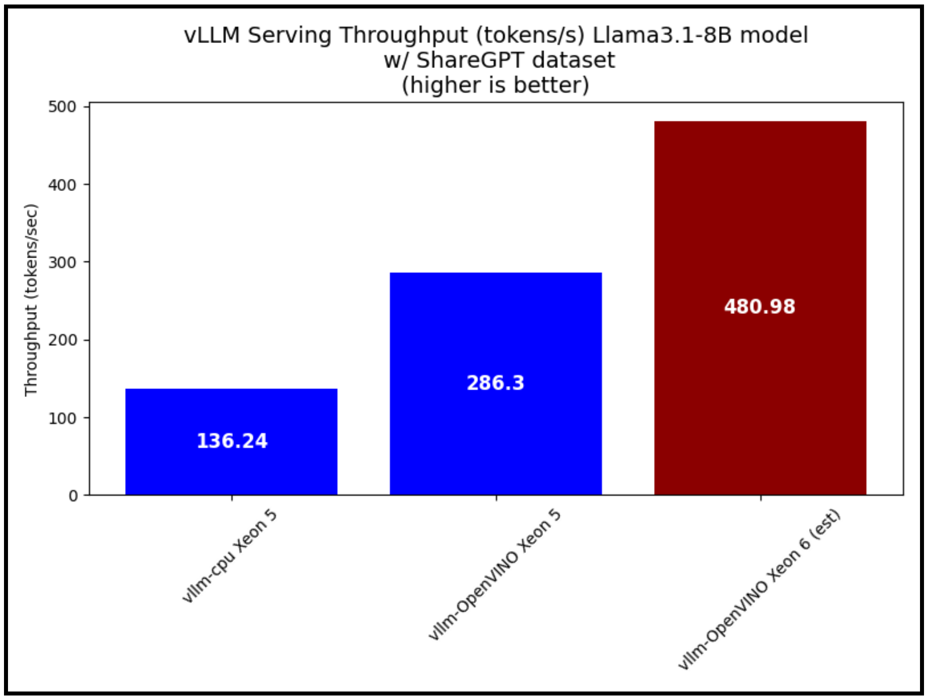

Figure 5. vLLM Serving Comparison (tokens per second)

Bill of Materials

The folllowing table provides a bill of materials for a cost-effective, CPU-only ThinkSystem SR650 V4 for AI Inferencing.

Table 1. Bill of Materials

Part number	Product Description	Qty
7DGDCTO1WW	**Server : ThinkSystem SR650 V4 - 3yr Warranty**	1
C3QK	ThinkSystem SR650 V4 24x2.5" Chassis	1
C3JB	ThinkSystem General Computing - Power Efficiency	1
BVGL	Data Center Environment 30 Degree Celsius / 86 Degree Fahrenheit	1
C5QY	Intel Xeon 6767P 64C 350W 2.4GHz Processor	2
C3QR	ThinkSystem V4 2U Performance Heatsink	2
C0TQ	ThinkSystem 64GB TruDDR5 6400MHz (2Rx4) RDIMM	8
5977	Select Storage devices - no configured RAID required	1
C0ZT	ThinkSystem 2.5" U.2 VA 7.68TB Read Intensive NVMe PCIe 5.0 x4 HS SSD	2
C46P	ThinkSystem 2U V4 8x2.5" NVMe Backplane	2
C0JK	ThinkSystem M.2 B340i-2i NVMe Enablement Adapter	1
BKSR	ThinkSystem M.2 7450 PRO 960GB Read Intensive NVMe PCIe 4.0 x4 NHS SSD	2
BE4U	ThinkSystem Mellanox ConnectX-6 Lx 10/25GbE SFP28 2-port PCIe Ethernet Adapter	1
C62D	ThinkSystem SR650/a V4 x16 Rear Direct Riser Slot 5	1
C0U3	ThinkSystem 2000W 230V/115V Titanium CRPS Premium Hot-Swap Power Supply	2
6400	2.8m, 13A/100-250V, C13 to C14 Jumper Cord	2
C3RQ	ThinkSystem V4 2U Standard Fan Module	6

Part number	**Product Description**	**Qty**
C2DH	ThinkSystem Toolless Slide Rail Kit V4	1
BQQ2	ThinkSystem 2U V3 EIA Latch Standard	1
BPKR	TPM 2.0	1
B7XZ	Disable IPMI-over-LAN	1
C3K9	XClarity Platinum Upgrade v3	1
C4S2	ThinkSystem SR650 V4 Processor board,BHS,DDR5,Santorini,2U	1
B0ML	Feature Enable TPM on MB	1
AVEQ	ThinkSystem 8x1 2.5" HDD Filler	1
AVEP	ThinkSystem 4x1 2.5" HDD Filler	1
AVEN	ThinkSystem 1x1 2.5" HDD Filler	10
BPP5	OCP3.0 Filler with screw	2
C3RM	ThinkSystem 2U Air duct Filler for 1P	2
AURS	Lenovo ThinkSystem Memory Dummy	24
C3RJ	ThinkSystem 2U 2LP Riser Cage Filler	2
C26Z	ThinkSystem GNR XCC CPU HS Clip	2
C3RN	ThinkSystem 2U Main Air Duct	1
C4SH	HV 2U V4 General WW L1 PKG BOM	1
C3S5	ThinkSystem 2U V4 3FH Riser Cage	1
BE05	Mellanox Low-Profile Dual-Port QSFP56 PCIe Bracket L1/SBB	1
C7Y8	ThinkSystem SR650 V4 System I/O Board	1
C3RH	ThinkSystem 2U 3FH Riser Cage Filler	1
C3QW	ThinkSystem M.2 Signal & Power Cable ,ULP 82P-SLX4/2X10 SB, 400/400mm	1
C3R0	ThinkSystem Power Cable, 2x6+12 P-2x3+6 P, 250 mm	2
C71U	Think System,PCIe Gen5 Cable, MCIOx8-MCIOx8, 250 mm	2
C6TH	Think System,PCIe Gen5 Cable, MCIOx8-MCIOx8, 350 mm	6
C3T9	ThinkSystem SR650 V4 model name Label	1
C3SQ	ThinkSystem SR650 V4 Agency label with ES&CE&UKCA	1
C20U	ThinkSystem 2000W TT power rating label WW	1
AWF9	ThinkSystem Response time Service Label LI	1
C3TH	ThinkSystem SR650 V4 Service Label for WW	1
B8K8	ThinkSystem 2U MS 24x2.5" NVMe HDD Type Label1	2
B97B	XCC Label	1
AUTQ	ThinkSystem small Lenovo Label for 24x2.5"/12x3.5"/10x2.5"	1
BQPS	ThinkSystem logo Label	1
BZ7F	ThinkSystem WW Lenovo LPK, Birch Stream	1
BE0E	N+N Redundancy With Over-Subscription	1
BK15	High voltage (200V+)	1
BTTY	M.2 NVMe	1
7S0XCTO8WW	**XClarity Controller Prem-FOD**	1
SCY0	Lenovo XClarity XCC3 premier - FOD	1
5641PX3	**XClarity Pro, Per Endpoint w/3 Yr SW S&S**	1

Part number	Product Description	Qty
1340	Lenovo XClarity Pro, Per Managed Endpoint w/3 Yr SW S&S	1
3444	Registration only	1

Intel Processor SKU list for AI Systems

The following table lists select Intel processor SKUs designed for use in AI systems, highlighting their core counts and generational differences to help guide hardware selection based on performance and workload requirements.

Table 2. Intel Processor SKU list for AI Systems

Family	Processor	Cores
4th Gen Intel Xeon (SPR)	8480C+	56
	8468	48
	8468V	48
	8462Y+	32
	6448H	32
5th Gen Intel Xeon (EMR)	8580	60
	8570	56
	8568Y+	48
	8562Y+	32
Intel Xeon 6 (GNR)	6960P	72
	6767P	64
	6761P	64
	6747P	48
	6740P	48
	6737P	32

References

For more information, see these resources:

- Docker Install
  https://docs.docker.com/engine/install/
- OpenVINO vLLM Plugin
  https://github.com/vllm-project/vllm-openvino
- Optimum-cli with OpenVINO options
  https://huggingface.co/docs/optimum/en/intel/openvino/export
- RAG Diagram
  https://en.wikipedia.org/wiki/Retrieval-augmented_generation
- Meta Blog Post - Methods for adapting large language models
  https://ai.meta.com/blog/adapting-large-language-models-llms/
- Meta Blog Post - To fine-tune or not to fine-tune
  https://ai.meta.com/blog/when-to-fine-tune-llms-vs-other-techniques/
- Meta Blog Post - How to fine-tune: Focus on effective datasets
  https://ai.meta.com/blog/how-to-fine-tune-llms-peft-dataset-curation/
- vLLM Command Line Arguments
  https://docs.vllm.ai/en/latest/serving/engine_args.html

Authors

Eric Page is an AI Engineer at Lenovo. He has 6 years of practical experience developing Machine Learning solutions for various applications ranging from weather-forecasting to pose-estimation. He enjoys solving practical problems using data and AI/ML.

Hapsara Sukasdadi is a seasoned IT and telecommunications industry expert. Serving as a Solutions architect, Hapsara currently drives Lenovo's AI and Telco technical engagements, focusing on architecting solutions in AI and Telco infrastructures. In this role, Hapsara collaborates closely with partners and ecosystem providers. Hapsara's primary mission is to deliver comprehensive solutions, encompassing design, planning, and integration across a spectrum of critical areas, including AI and Telecommunications infrastructure solutions.

Related product families

Product families related to this document are the following:

- Artificial Intelligence
- Processors

Notices

This document, LP2249, was created or updated on July 1, 2025.

Send us your comments in one of the following ways:

- Use the online Contact us review form found at:
  https://lenovopress.lenovo.com/LP2249
- Send your comments in an e-mail to:
  comments@lenovopress.com

This document is available online at https://lenovopress.lenovo.com/LP2249.

Trademarks