Solution White Paper

Artificial Intelligence

Optimizing Large Language Models with the OpenVINO™ Toolkit

Authors

Ria Cheruvu

Intel AI Evangelist

Ryan Loney

Intel OpenVINO Product Manager

Credits: Ekaterina Aidova, Alexander Kozlov, Helena Kloosterman, Artur Paniukov, Dariusz Trawinski, Ilya Lavrenov, Nico Galoppo, Jan Iwaszkiewicz, Sergey Lyalin, Adrian Tobiszewski, Dariusz Trawinski, Jason Burris, Ansley Dunn, Michael Hansen, Raymond Lo, Yury Gorbachev, Adam Tumialis, and Milosz Zeglarski.

Table of Contents

Executive Summary

Large language models (LLMs) have enabled breakthroughs in natural language understanding, conversational AI, and diverse applications such as text generation and language translation. Additionally, large language models are massive, often over 100 billion parameters and growing. A recent study published in Scientific American[1] by Lauren Leffer articulates the challenges with large AI models, including scaling to small devices, accessibility when disconnected from the internet, as well as power consumption and environmental concerns. This solution white paper describes methods to optimize large language models though compression techniques. The OpenVINO™ toolkit is a leading solution for optimizing and deploying LLMs in end-user systems and devices. Developers use OpenVINO™ to compress LLMs, integrate them into AI-assistant applications, and deploy them on edge devices or on the cloud with maximum performance.

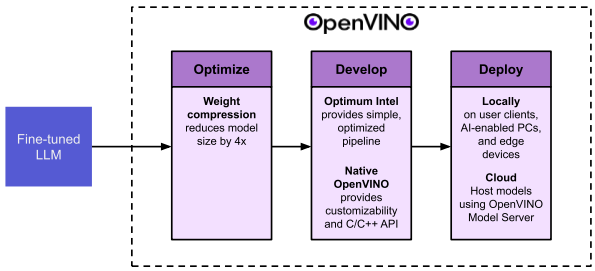

Figure 1. Developer Workflow

The OpenVINO™ Deployment Advantage

The OpenVINO™ toolkit offers the following advantages over other LLM deployment solutions:

- **Small Binary Size:** OpenVINO™ requires fewer dependencies and a smaller footprint than Hugging Face*, PyTorch*, and other machine learning frameworks, allowing it to have a smaller deployment footprint.
- **Speed:** OpenVINO™ provides optimized inference for LLMs and is one of the only runtime libraries that provides a full C/C++ API for efficient LLM inference, suitable for high-performance, resource-optimized production environments, in addition to a Python* API.
- **Official Intel Support:** OpenVINO™ is the official AI framework distributed by Intel, and it will be fully supported with patches, monthly releases, and feature updates from Intel engineers.

This solution white paper explains the benefits of using OpenVINO™ for LLM deployment. It gives comprehensive information, examples, and code snippets showing how to use OpenVINO™ to compress LLMs and build text generation pipelines around them. It also provides examples showing how to deploy LLMs on OpenVINO™ Model Server or in a chatbot application. Read on to learn more about OpenVINO's capabilities with LLMs.

A brief introduction to LLMs

Large language models (LLMs) are massive neural networks built on transformers that excel in various language-related tasks like text completion, question-answering, and content summarization. They have become hugely popular with the release of OpenAI's ChatGPT, an "AI assistant" centered around a state-of-the-art LLM known as GPT-4.

This solution white paper focuses on causal LLMs, which are LLMs that focus on predicting the next word in a sequence and are the type most used for AI chatbots.

Training LLMs

LLMs are trained from an extensive repository of text data, which is usually collected by crawling web pages across the internet. LLMs are trained for a simple task: given a sequence of words, predict the next word in the sequence. To accomplish this task, the billions of parameters in the LLM's transformer network gradually adjust during training until they can accurately predict the content of the text documents in the training data. Essentially, this training process compresses the entirety of publicly available text on the internet (> 10TB of data) into a neural network of parameters (1GB - 200GB of data).

Training LLMs from scratch is an expensive process! Millions of GPU hours are required for the model to fit to the training data. Meta's* Llama 2 models were trained using over 3 million GPU hours at an estimated cost of $2,000,000 USD (reference[2]). Training from scratch is typically only done by research groups in large companies. Fortunately, many trained models are released to the public under open-source licenses so they can be fine-tuned to a unique application.

Fine-tuning LLMs

Fine-tuning a large language model allows it to adapt its pre-trained knowledge to specific tasks. While the model may be good at predicting words in a document, it is not yet adapted to providing human-like responses to questions. It is only trained on publicly available data up to a specific cut-off date, so it may not have knowledge of current events. In the fine-tuning stage, a new training dataset is used that consists of a smaller number of high-quality text examples that are often hand-selected or generated by humans. Fine tuning aligns the model to this narrower dataset related to the target task so it can generate more accurate responses.

For example, to convert the Llama 2 base model into a question-answering AI assistant, Meta researchers fine-tuned it using a dataset of about 100,000 samples of hand-picked documents or human-written responses to prompts. Similarly sized datasets may be used to align an LLM to provide more customized responses and improve its knowledge on specific topics.

Fine-tuning a LLM requires much less resources than training an LLM from scratch. Parameter efficient fine tuning (PEFT) techniques such as Low Rank Adaptation (LoRA)[3] and QLoRA reduce the memory requirements. With these techniques, fine-tuning can be accomplished on a computer equipped with a high-end GPU. Fine-tuning Llama 2-7B using Hugging Face's PEFT LoRA method takes about 16 hours on a single GPU and uses less than 10GB GPU memory.

Current popular LLMs

The field of generative AI research moves fast, and state-of-the-art models are released on a monthly or even weekly basis. Here is a list of popular open source chatbot LLMs at the time of March 2024.

- Baichuan 2 7B (Baichuan Intelligent Technology*)
- ChatGLM3 6B (Tsinghua* University)
- Llama 2 family: Llama 2-7B, Llama 2-13B, and Llama 2-70B (Meta*)
- Mistral 7B (Mistral* AI)
- Mixtral-8x7B (Mistral AI)
- Qwen 7B (Alibaba*)
- StableLM 7B (Stability AI*)
- Vicuna 7B (LMSYS*)
- Zephyr 7B (Hugging Face)

These LLMs are all supported by OpenVINO™, along with additional versions of these models with larger weights.

Note: OpenAI's* GPT-4 model, and Anthropic's* Claude 2 model outperform all the open-source models listed above in terms of response accuracy and quality. However, both models are closed source and not available for fine-tuning.

System requirements for running LLMs

One limitation of LLMs is their size: a 70-billion parameter model like Llama 2-70B takes around 140GB worth of space. The system must have enough storage space and memory to hold the model's weights.

- Storage requirements: The system must have enough disk space to store the model files. The total file size of LLMs can range from 2GB for small models to over 300GB for large models.

- RAM requirements: The amount of RAM used during inference depends on the context length, model size, and other factors. As a rule of thumb, the system should have at least as much RAM as the size of the model file. If it doesn't have enough RAM available to load the full model, the system may resort to using disk storage as swap space for memory, which will cause inference to run slowly. Or it may just crash when memory is exhausted.

- GPU vRAM requirements (if running on GPU): The GPU must have at least as much vRAM as the size of the model file with sufficient memory padding for other OS tasks.

Fortunately, these requirements can be significantly reduced using weight compression. Compressing the model's weights from FP32 to INT8 quarters its size and converting to INT4 format leads to approximately 1/8th of its size. For example, the Zephyr-7B-beta model in FP32 format has a file size of 28 GB. When it is compressed to INT4 using OpenVINO™ NNCF, it reduces the file size to 4GB while maintaining similar accuracy. To learn more, see the *Weight Compression With OpenVINO™* section of this document.

Generally, a computer that has a mid-to-high-end processor and 16GB of RAM can safely run quantized 7B models such as Llama-7B or Zephyr-7B-beta. These models provide a good entry point to experiment with text generation pipelines. As models increase in size and number of parameters, higher-end hardware is needed to support them.

Why use OpenVINO™ for LLMs

OpenVINO™ provides a flexible and efficient runtime for deploying LLMs. Its advantages include deployment size, speed, support, flexibility, and ability to run on a variety of hardware.

Slim Deployment

OpenVINO™ is a self-contained package that requires fewer dependencies than Hugging Face, PyTorch, and other machine learning frameworks. Hugging Face and PyTorch environments require several gigabytes worth of dependencies, while OpenVINO™ only requires several hundred megabytes.

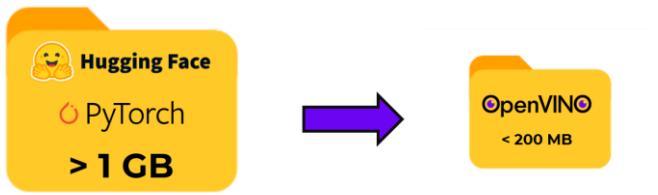

Figure 2. Benefit of OpenVINO™ is its reduced footprint size compared to other frameworks

The slimmer binary size and memory footprint of OpenVINO™ reduce the storage requirements for target hardware and make containers easier to deploy and update. Fewer dependencies mean less headache with package and version management for deployment environments.

Speed

OpenVINO™ provides optimized inference for LLMs, and it is constantly being improved for even faster performance. Solutions built with OpenVINO™ are as fast or faster than other third-party solutions such as the open source llama.cpp distribution.

Most other LLM-capable runtimes rely on Python code executing through a Python interpreter. OpenVINO™ is one of the only runtime libraries that provides a full C/C++ API for inference with LLMs, targeted for resource-optimized production environments. OpenVINO™ allows LLM applications to be built and optimized for target processors using C/C++.

Of course, OpenVINO™ also offers a Python API, which allows for quicker development of algorithms and programs: Prototype a solution in Python, and then optimize it in C++ using OpenVINO™.

See the OpenVINO Performance Benchmarks page for graphs showing latency and throughput of LLMs on various platforms. For more information on how to benchmark LLMs, see the *Benchmarking LLMs and Measuring Accuracy* section of this solution white paper.

Below, we include a snapshot of benchmarking data on a few, select CPU-only and GPU Intel hardware platforms:

Table 1 shows benchmark latency (milliseconds per token) and throughput (tokens per second) for three Intel processors with three generative AI models, chatGLM2-6b, Llama-2-7b-chat, and Mistral-7b, with FP16, INT8, and INT4 weights using OpenVINO version 2024.0.

HW Platforms:	Model Name:	Throughput (tokens/sec) Tokens Input: 1024 Tokens Output: 128 Beam: 1				Latency (msec/token) Tokens Input: 1024 Tokens Output: 128 Beam: 1			
		INT4	INT8	FP32	BF16	INT4	INT8	FP32	BF16
Intel® Core™ i9-13900K CPU-only 24 Cores, Memory: 2 x 32 GB DDR5 4800MHz	chatGLM2-6b	9.5	7.4	2.5	N/A	104.74	134.86	405.05	N/A
	Llama-2-7b-chat	8.9	6.7	2.2	N/A	112.65	148.81	458.75	N/A
	Mistral-7b	9.4	6.3	2.0	N/A	106.24	157.95	491.22	N/A
Intel® Xeon® Platinum 8380 CPU-only 40 Cores, Memory: 16 x 16 GB DDR4 3200MHz	chatGLM2-6b	12.6	9.2	4.7	N/A	79.25	108.13	211.30	N/A
	Llama-2-7b-chat	17.4	10.4	3.6	N/A	57.52	95.73	280.26	N/A
	Mistral-7b	12.7	7.5	3.7	N/A	78.61	133.73	267.73	N/A
Intel® Xeon® Platinum 8490H CPU-only 60 Cores, Memory: 16 x 16 GB DDR5 4800MHz	chatGLM2-6b	31.5	23.4	N/A	15.7	31.79	42.70	N/A	63.66
	Llama-2-7b-chat	27.0	20.1	N/A	13.6	37.1	49.75	N/A	73.68
	Mistral-7b	28.2	19.2	N/A	12.7	35.43	51.95	N/A	78.85

Table 1. Generative AI benchmarks (latency and throughput) on Intel CPU-only Processors

Table 2 shows benchmark latency (milliseconds per token) for GPU-based Intel processors (both discrete and integrated GPU) with the three generative AI models, chatGLM2-6b, Llama-2-7b-chat, and Mistral-7b, with INT4 weights using OpenVINO version 2024.0.

HW Platforms:	Model Name:	Throughput (tokens/sec) Tokens Input: 1024 Tokens Output: 128 Beam: 1			Latency (msec/token) Tokens Input: 1024 Tokens Output: 128 Beam: 1		
		INT4	INT8	FP32	INT4	INT8	FP32
Intel® Core™ Ultra 7 Processor 165H iGPU	chatGLM2-6b	Not measured	Not measured	4.2	Not measured	Not measured	240.22
	Llama-2-7b-chat	8.5	5.4	3.3	116.97	183.6	305.68
	Mistral-7b	6.0	4.3	2.5	166.45	231.75	393.34
Intel® Data Center GPU Flex 170 dGPU	chatGLM2-6b	Not measured	Not measured	Not measured	Not measured	Not measured	Not measured
	Llama-2-7b-chat	12.0	15.3	Not measured	83.57	65.31	Not measured
	Mistral-7b	9.9	13.3	Not measured	101.29	75.05	Not measured
Intel® Arc™ A-Series Graphics A770M dGPU	chatGLM2-6b	Not measured	Not measured	11.8	Not measured	Not measured	85.09
	Llama-2-7b-chat	8.6	10.3	10.7	116.82	97.45	93.26
	Mistral-7b	6.2	7.5	7.1	162.26	132.71	141.58
Intel® Core™ i7-1360P Processor iGPU	chatGLM2-6b	Not measured	Not measured	4.2	Not measured	Not measured	240.22
	Llama-2-7b-chat	4.6	3.5	Not measured	217.08	286.83	Not measured
	Mistral-7b	3.2	Not measured	Not measured	309.56	Not measured	Not measured

Table 2. Generative AI benchmarks (latency) on Intel GPU Processors

Official Intel support

While several open-source solutions exist for deploying LLMs in C/C++ applications, they are created from a community-based effort and may or may not be supported over the long term. OpenVINO™ is the official AI framework distributed by Intel, and it will be fully supported with patches, upgrades, and feature updates from Intel engineers. Software vendors and OEMs building LLM applications around OpenVINO™ have access to Intel field application engineers and the Intel Developer Software Forums to answer questions and debug errors.

OpenVINO™ is developed and maintained on a monthly release schedule. Every month, new features and patches will be shipped to keep OpenVINO™ up to speed with the fast-moving field of AI and LLMs. As new technology advancements occur, other community-developed libraries may not release patches to support those advancements.

Flexibility

OpenVINO™ is a flexible and efficient library for developing and deploying AI applications. OpenVINO's flexibility makes it simple to import and run deep learning models from all popular frameworks. While specialized deployment solutions like llama.cpp only support LLMs, OpenVINO™ supports all kinds of models and architectures. This enables development of multimodal applications ranging from computer vision, image generation, text-to-speech, data classification, and much more.

Training frameworks such as PyTorch also have great flexibility for developing and deploying deep learning models, but they are not as optimized and efficient as OpenVINO™. The C/C++ APIs in OpenVINO™ provide an inherent advantage over the Python APIs of other frameworks. OpenVINO™ allows developers to write an application once and deploy it anywhere, with maximum performance from hardware.

Hardware support

OpenVINO™ supports LLM deployment on a wide range of hardware devices, including CPUs, integrated GPUs and discrete GPUs. It supports ARM-based architectures as well as x86/x64 architectures. This range of hardware support enables LLMs to be deployed on a wide variety of targets, ranging from high-powered servers to compact edge devices.

OpenVINO's automated optimization squeezes a maximum amount of performance out of the target hardware without needing to reconfigure the application. On newer hardware (such as Intel® Advanced Matrix Extensions (Intel® AMX) -enabled products like our newer **Intel® Xeon®** processors), OpenVINO™ automatically selects the best data type for inference. For more information, see the "Optimize Inference" and "Precision Control" pages for additional information, including how INT8 quantized models are run by default with BF16 plus INT8 mixed precision, taking full advantage of the AMX capability of 4th Generation Intel® Xeon® Scalable Processors.

How to optimize and deploy with OpenVINO™

There are two options for optimizing and deploying LLMs with OpenVINO™:

1. **Hugging Face:** Use OpenVINO™ as a backend for the Hugging Face Transformers API through the Optimum Intel extension.
2. **Native OpenVINO™:** Use OpenVINO native APIs (Python and C++) with custom pipeline code.

In both cases, the OpenVINO™ runtime is used as the backend for inference, and OpenVINO™ tools are used for model optimization. The main differences are in ease of use, footprint size, and customizability.

The Hugging Face API is easy to learn and provides a simpler interface for the developer. It hides the complexity of model initialization and text generation through high-level methods and classes. However, it has more dependencies, provides abstractions of the text generation loop, scheduler, tokenizer, and other elements of the LLM workflow leading to less options for detailed customization, and cannot be ported to C/C++.

The native OpenVINO™ API requires fewer dependencies, minimizing the size of the application footprint, and can be used to build efficient C/C++ applications. However, it has a steeper learning curve, and it requires explicit implementation of the text generation loop, tokenization functions, and scheduler functions used in a typical LLM pipeline.

Category	Hugging Face* with Optimum Intel	Native OpenVINO™
Ease of use	Lower learning curve; quick to integrate	Higher learning curve; requires more effort in integration
Library dependencies	Many Hugging Face dependencies	Lightweight (e.g., numpy, etc.)
Languages supported	Python	Python, C/C++
Ideal use case	Ideal for Python-centric projects	Best suited for high-performance, resource-optimized production environments

Table 3. Comparing Hugging Face and Native OpenVINO™

Hugging Face's libraries enable experimenting with different models to build out applications. For leveraging further optimizations, the model and application can be ported to OpenVINO™ using OpenVINO™ APIs.

This Solution White Paper shows how to optimize and deploy LLMs using both Hugging Face and native OpenVINO™. It begins with a Quick Start example using Hugging Face, and then provides more details on how to load LLMs, compress them, and run inference.

Install Dependencies

To get started with OpenVINO™, set up a Python virtual environment for OpenVINO™ by following the OpenVINO Installation Instructions.

Once the environment is created and activated, install Optimum Intel, OpenVINO™, NNCF and their dependencies in a Python environment by issuing:

```
pip install optimum[openvino]
```

Quick start LLM inference example

The example below shows a quick way to get started with LLMs and see the basics of how they work. It does the following:

1. It loads the zephyr-7b-beta[4] LLM from Hugging Face using the Optimum Intel API, which converts it to OpenVINO™ Intermediate Representation (IR) format and sets OpenVINO™ as the backend for inference.
2. It automatically compresses the model to INT8 format using OpenVINO™ NNCF by default.
3. It loads a tokenizer for converting an input text prompt into tokens that can be understood by the model.
4. It sets up an inference pipeline with the model and tokenizer, passes in an input prompt, and prints the resulting response.

By default, this example runs the model on CPU, but the GPU can be used instead by uncommenting the `model.to("GPU")` line.

```
# Import necessary packages
from optimum.intel import OVModelForCausalLM
from transformers import AutoTokenizer, pipeline

# Load zephyr-7b-beta model and tokenizer from Hugging Face
model_id = "HuggingFaceH4/zephyr-7b-beta"
model = OVModelForCausalLM.from_pretrained(model_id, export=True)
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Optional: compile model to run on GPU
#model.to("GPU")

# Set up pipeline and perform inference
pipe = pipeline("text-generation", model=model, tokenizer=tokenizer, max_length=50)
prompt = "My cat's favorite foods are"
results = pipe(prompt)
print(results)
```

Here's the output from running the example. Try changing the prompt to see what other text outputs can be generated!

```
"My cat's favorite foods are wet cat food, especially those with gravy or pate, and dry cat food
with chicken or fish flavors. She also enjoys treats with chicken or tuna flavors."
```

The above example hides much of the complexity involved with LLM inference behind the high-level `pipeline` and `AutoTokenizer` classes. These are great for setting up simple examples, but they don't allow for much customization. Also, they are Python classes, so they can't be used in C/C++ applications. Let's dive into more details about how to create and customize LLM applications in OpenVINO™.

Loading LLMs into OpenVINO™

Loading Hugging Face LLMs into OpenVINO™

The easiest way to use a LLM in OpenVINO™ is to load a model from the Hugging Face Hub using Optimum Intel. Models loaded with Optimum Intel are optimized for OpenVINO™ while being compatible with the Hugging Face Transformers API. The `OVModelForCasualLM` class takes a model name, downloads it from Hugging Face*, and initializes it as an object in memory.

To initialize a model from Hugging Face, use the `OVModelForCasualLM.from_pretrained` method as shown in the snippet below. By setting the parameter `export=True`, the model is converted to OpenVINO™ IR format on the fly.

```
from optimum.intel import OVModelForCausalLM
model_id = "HuggingFaceH4/zephyr-7b-beta"
model = OVModelForCausalLM.from_pretrained(model_id, export=True)
```

Saving and Loading Models

Once a model has been converted to IR format using Optimum Intel, it can be saved and exported to use in a future session or in a deployment environment. The conversion process takes a while, so it's preferable to convert the model to IR format once, save it, and then load the compressed model later for faster time to first inference.

To save and export a model and its tokenizer, use `model.save_pretrained("your-model-name")` and `tokenizer.save_pretrained("your-model-name")` as shown in the snippet below.

```
# Save model for faster loading later
model.save_pretrained("zephyr-7b-beta-ov")
tokenizer.save_pretrained("zephyr-7b-beta-ov")
```

The model will be exported in OpenVINO™ IR format (`openvino_model.bin`, `openvino_model.xml`) and saved to a new folder in the specified directory. The tokenizer will also be saved to the directory.

To load the model and tokenizer in a future session, use `OVModelForCausalLM.from_pretrained("your-model-name")` and `AutoTokenizer.from_pretrained("your-model-name")` as shown in the snippet below.

```
# Load a saved model
model = OVModelForCausalLM.from_pretrained("zephyr-7b-beta-ov")
tokenizer = AutoTokenizer.from_pretrained("zephyr-7b-beta-ov")
```

Converting a Hugging Face Model to OpenVINO™ IR using CLI

Another way to convert models from Hugging Face to OpenVINO™ IR format is using the **optimum-cli** tool. This is helpful for converting models without using a Python script.

The command to perform this conversion is structured as follows:

```
optimum-cli export openvino --model <MODEL_NAME> <NEW_MODEL_NAME>
```

`--model <MODEL_NAME>`: This part of the command specifies the name of the mode to be converted. Replace `<MODEL_NAME>` with the actual model name from Hugging Face.

`<NEW_MODEL_NAME>`: Here, you specify the name you want to give to the new model in the OpenVINO™ IR format. Replace `<NEW_MODEL_NAME>` with your desired name.

For example, to convert the Llama 2-7B model from Hugging Face (formally named as `meta-llama/Llama-2-7b-chat-hf`) to an OpenVINO™ IR model and name it "ov_llama_2", use the following command:

```
optimum-cli export openvino --model meta-llama/Llama-2-7b-chat-hf ov_llama_2
```

In this example, **meta-llama/Llama-2-7b-chat-hf** is the Hugging Face model name, and **ov_llama_2** is the new name for the converted OpenVINO™ IR model.

Additionally, you can specify the --weight-format argument to apply 8-bit or 4-bit weight quantization when exporting your model with the CLI. An example command applying 8-bit quantization to the model **gpt2** is below:

```
optimum-cli export openvino --model gpt2 --weight-format int8 ov_gpt2_model
```

Special Case: Loading models tuned with LoRa

Low-rank adaptation (LoRA) is a popular method to tune generative AI models to a downstream task or custom data. With LoRA, smaller representations of the model weights (called weight adapters) are trained to adapt to new data without needing to adjust the entire model. To learn more about LoRA, see the LoRA article[5] from Hugging Face.

Models that have been fine-tuned using LoRA require a few extra steps when being loaded. The trained weight adapters (which are produced during LoRA training) must be merged into the baseline model using the `merge_and_unload()` function before the model is used for inference. For example:

```
model_id = "meta-llama/Llama-2-7b-chat-hf"
lora_adaptor = "./lora_adaptor"
model = AutoModelForCausalLM.from_pretrained(model_id, use_cache=True)
model = PeftModelForCausalLM.from_pretrained(model, lora_adaptor)
model.merge_and_unload()
model.get_base_model().save_pretrained("fused_lora_model")
```

Now the model can be converted to OpenVINO™ using the Optimum Intel API or CLI options mentioned above.

Weight compression with OpenVINO™

Weight compression is a technique for reducing the size of LLMs. It significantly reduces the memory required to store the model's weights during inference, thereby reducing the amount of system RAM or vRAM needed to run the model. The OpenVINO™ Neural Network Compression Framework (NNCF) provides tools for performing weight compression on LLMs.

Unlike full model quantization, where weights and activations are quantized, weight compression in NNCF only targets the model's weights. This approach allows the activations to remain as floating-point numbers, preserving most of the model's accuracy. It's a subtle yet important difference that ensures the accuracy of the model is maintained while improving its speed and reducing its size. The reduction in size is especially noticeable with larger models. For instance, the Llama-2 7B model can be reduced from about 25GB to 4GB using 4-bit weight compression.

Weight compression should be performed offline rather than in a real-time application. The LLM can be compressed and exported in a development environment and then used in a deployment environment.

Benefits of weight compression

LLMs and other models that require extensive memory to store the weights during inference can benefit from weight compression in the following ways:

- Enables inference of exceptionally large models that cannot otherwise be accommodated in the device memory
- Reduces storage and memory overhead by implementing model weight compression, making models more lightweight and less resource intensive for deployment
- Improves inference speed by reducing the latency of memory access when calculating outputs of each layer (the weights are smaller and thus faster to load from memory)
- Unlike full quantization, weight compression does not require sample data to calibrate the range of activation values, so it is easier to perform

Weight compression data types

NNCF supports three types of LLM weight compression:

- INT8: 8-bit weight quantization
- INT4_SYM: 4-bit symmetric weight quantization
- NT4_ASYM: 4-bit asymmetric weight quantization

The following sections explain the differences between each option.

INT8 – 8-bit weight quantization

The default compression method is the INT8 mode, which compresses weights to an 8-bit integer data type. This mode offers a balance between model size reduction and maintaining accuracy, making it a versatile option for a broad range of applications. It significantly reduces the model size compared to higher-bit formats while ensuring that the accuracy of the model remains similar. This makes INT8 an ideal starting point for many models.

INT4_SYM - 4-bit symmetric weight quantization

INT4 symmetric mode involves quantizing weights to an unsigned 4-bit integer symmetrically around a fixed zero point of 8 (i.e., the midpoint between 0 and 15). Inference with a model compressed using this mode is faster than a model with INT8 precision, making it ideal for situations where speed is prioritized over accuracy. Although it may lead to some degradation in accuracy, it is well-suited for models where this trade-off is an acceptable exchange for a noticeable gain in speed and reduction in size.

INT4_ASYM - 4-bit asymmetric weight quantization

INT4 asymmetric mode also quantizes weights to unsigned 4-bit integers but does so asymmetrically with a non-fixed zero point. This mode slightly compromises speed in favor of better accuracy compared to the symmetric mode. This mode is useful when minimal accuracy loss is crucial, but a faster performance than INT8 is desired.

Comparison table

Table 4 summarizes the benefits and trade-offs for each compression type in terms of memory reduction, speed gain, and accuracy loss.

	Memory Reduction	Latency Improvement	Accuracy Loss
INT8	Low	Medium	Low
INT4 Symmetric	High	High	High
INT4 Asymmetric	High	Medium	Medium

Table 4. Comparing compression types

Performing weight compression with Optimum Intel

The example below shows how to perform weight compression on a model from Hugging Face. In the example, a Zephyr-7B-beta model is loaded from Hugging Face using Optimum Intel. When the model is loaded, it is automatically compressed to the specified compression type using NNCF.

The compression type is specified in the `OVModelForCausalLM.from_pretrained` method using the `compression_option="<option>"` argument. When this option is set to none, INT8 weight compression will by enabled by default for decoder models with more than 1B parameters. It accepts any of the following options:

- `"int8"` : INT8 compression using NNCF
- `"int4_sym_g128"` : Symmetric INT4 compression with group size of 128
- `"int4_asym_g128"` : Asymmetric INT4 compression with group size of 128
- `"int4_sym_g64"` : Symmetric INT4 compression with group size of 64
- `"int4_asym_g64"` : Asymmetric INT4 compression with group size of 64

For more information on group size, see the Weight Compression page in OpenVINO™ documentation. In this example, `compression_option="int8"` is used to indicate INT8 quantization should be performed.

```
from nncf import compress_weights, CompressWeightsMode
from optimum.intel.openvino import OVModelForCausalLM
from transformers import AutoTokenizer, pipeline

# Load model from Hugging Face and compress to INT8
model_id = HuggingFaceH4/zephyr-7b-beta"
model = OVModelForCausalLM.from_pretrained(model_id, export=True, compression_option="int8")

# Inference
tokenizer = AutoTokenizer.from_pretrained(model_id)
pipe = pipeline("text-generation", model=model, tokenizer=tokenizer)
phrase = "The weather is"
results = pipe(phrase)
print(results)

# Save compressed model and tokenizer for later use
model.save_pretrained("zephyr-7b-beta-int8-sym-ov")
tokenizer.save_pretrained("zephyr-7b-beta-int8-sym-ov")
```

Note at the end of the example, the compressed model and its tokenizer are saved so they can be imported for use in a future session. This saves the time of compressing the model whenever it is used in a new session. For more information, see the *Saving and Loading Models* section.

Performing weight compression with Neural Network Compression Framework (NNCF)

The example below shows how to perform weight compression on an OpenVINO™ IR model using NNCF. In the example, a model in OpenVINO™ IR format is read using `ov.core.read_model`. The `nncf.compress_weights` method is used to quantize the model weights to the specified data type.

The compression type used by the `nncf.compress_weights` method is set using the `mode` argument, which accepts one of the three following options:

- `mode=CompressWeightsMode.INT8`
- `mode=CompressWeightsMode.INT4_SYM`
- `mode=CompressWeightsMode.INT4_ASYM`

In this example, the INT4_SYM mode is set to use 4-bit symmetric quantization.

```
from nncf import compress_weights, CompressWeightsMode
import openvino as ov

# Read an OpenVINO IR model
core = ov.Core()
model = core.read_model("model.xml")

# Compress to INT4 Symmetric
model = compress_weights(model, mode=CompressWeightsMode.INT4_SYM)

# Save compressed model for later use
ov.save_model(model, "model-int4-sym.xml")
```

NNCF also allows for configuring the `group_size` and `ratio` compression parameters, which can be used to tweak the size and inference speed of the compressed model. For more information on these parameters, see the Weight Compression page in OpenVINO™ documentation.

LLM inference with OpenVINO™

At the core of a text generation application such as a chat bot, a LLM runs inference on a set of inputs to produce a set of outputs. The inputs, in this case, are a sequence of words that have been converted to tokens. The outputs are a set of candidate tokens and their probability of being the next token in the sequence. The application selects the best candidate token and appends it to the input sequence. This process repeats in a loop until a maximum sequence length is reached or an "end of sequence" token is generated, and the resulting sequence is detokenized and displayed to the user.

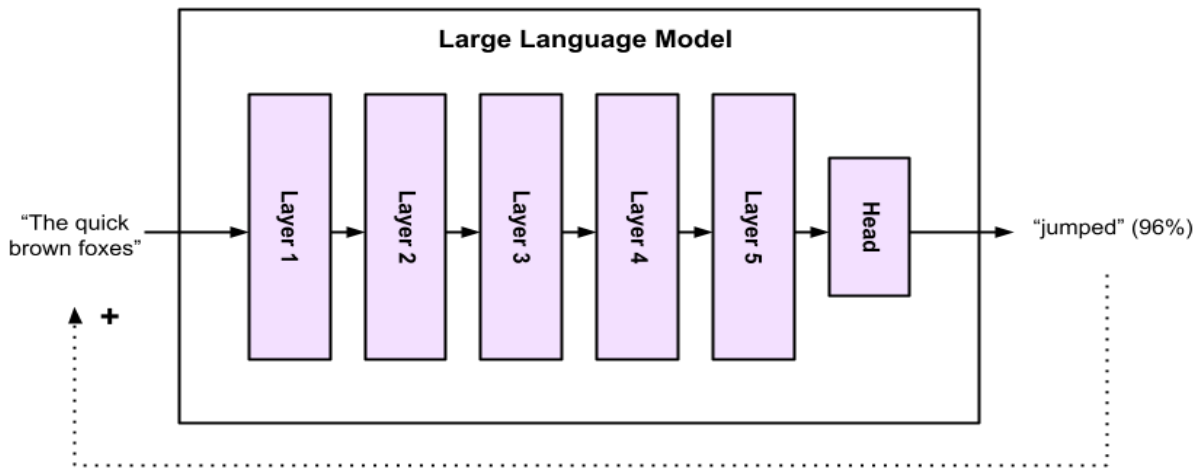

Figure 3. To generate text, LLM predicts the next best word in the input sequence, appends it to the input, and repeats the process until an "end of sequence" token is generated.

The token generation inference loop is one stage of a text generation application. The other stages are loading the model, tokenizing the input text, and processing the output tokens to be displayed to the user. The four stages are listed in order below:

1. Load model
2. Tokenize input text
3. Execute inference loop
4. Process output tokens

These stages look very different depending on if Hugging Face or the native OpenVINO™ API is used for implementation. This section of the Solution White Paper shows how to implement LLM text generation using both approaches.

Inference with Hugging Face and Optimum Intel

Hugging Face's high-level Transformers API provides a simple option for initializing the model and running inference. It is wrapped with the Optimum Intel extension, which converts the LLM to OpenVINO™ IR format and sets OpenVINO™ runtime as the backend.

Shown below is the Quick Start example given earlier in this document that uses Hugging Face Transformers and Optimum Intel to set up and run a simple text generation pipeline.

Basic example :

```python
from optimum.intel import OVModelForCausalLM
# new imports for inference
from transformers import AutoTokenizer, pipeline


# load the model
model_id = "meta-llama/Llama-2-7b-chat-hf"
model = OVModelForCausalLM.from_pretrained(model_id, export=True)


# inference
tokenizer = AutoTokenizer.from_pretrained(model_id)
pipe = pipeline("text-generation", model=model, tokenizer=tokenizer)
prompt = "What is OpenVINO?"
results = pipe(prompt)
```

Basic example stages:

Stage	API & Method Used
1. Load model	Optimum Intel from_pretrained
2. Tokenize inputs	Transformers AutoTokenizer*
3. Execute inference loop	Optimum Intel generate*
4. Process output tokens	Transformers AutoTokenizer*

*Used inside transformers.pipeline class

Figure 4. A basic text generation example using Hugging Face Transformers and Optimum Intel.

In the example, three key classes and methods are used:

- `OVModelForCausalLM.from_pretrained` from Optimum Intel: Loads the LLM from Hugging Face, converts it to OpenVINO™ IR format, and compiles it on a target device using OpenVINO™ as the inference backend
- `AutoTokenizer` from Hugging Face Transformers: Initializes a text tokenizer for the LLM
- `Pipeline` from Hugging Face Transformers: Handles the bulk of text generation, including tokenizing the inputs, executing the inference loop, and processing the outputs

These classes provide a simple interface for setting up text generation. Each class and method has more parameters that can be used to further configure the model or the text generation process. The Transformers API also has other features that give more control over inference parameters, such as the model.generate() method. To learn more, visit the Hugging Face Transformers documentation[6] page.

There are two options to select which device (CPU, iGPU, GPU, etc) the LLM is compiled on:

1. Specify the `device` parameter in the `.from_pretrained()` call. For example, use `OVModelForCausalLM.from_pretrained (model_id, export=True, device="GPU.0")` to run the model on the GPU. See the Device Query documentation for more information on how target devices are named and enumerated.
2. Use the `model.to` method after the model has been loaded and pass in the name of the target device. For example, use `model.to("GPU.0")` to run the model on the GPU.

While the Hugging Face APIs greatly simplify the code for implementing text generation, one drawback is that they cannot be implemented in C/C++. In contrast, the native OpenVINO™ API supports building solutions with C/C++.

Inference with Native OpenVINO™ API in Python

Inference can also be run on LLMs using the native OpenVINO™ API. An inference pipeline for a text generation LLM can be set up in the following stages:

1. Read and compile the model
2. Tokenize text and set model inputs
3. Run token generation loop
4. De-tokenize outputs

This section provides code snippets showing how to implement each stage with the native OpenVINO Python API. These snippets implement a stateful model technique to increase the memory efficiency of LLMs. With this technique, the context of the model, i.e. its internal states (the KV cache), are shared among multiple iterations of inference: The KV cache that belongs to a particular text sequence is accumulated inside the model during the generation loop. The stateful model implementation supports both greedy search and beam search (preview) for LLMs. This technique also reduces the memory footprint of LLMs, for example for being able to run INT4 models.

Prerequisites: Install OpenVINO™ Tokenizers

The OpenVINO Tokenizers module will be used for tokenization. It is supported on Linux, macOS, and Windows operating systems. An updated list of supported tokenizer types can be found in the "Supported Tokenizer Types" section of the OpenVINO™ Tokenizers documentation.

In the same Python virtual environment that was set up in the *Install Dependencies* section of this Solution White Paper, install OpenVINO™ Tokenizers by issuing:

```
pip install openvino-tokenizers[transformers]
```

Convert Hugging Face Tokenizer and Model to OpenVINO™ IR Format

Before an LLM and its tokenizer can be used with the native OpenVINO™ API, it must be converted to OpenVINO™ IR format.

OpenVINO™ Tokenizer comes equipped with a CLI tool, `convert_tokenizer`, that converts tokenizers from the Hugging Face Hub to OpenVINO™ IR format:

```
convert_tokenizer HuggingFaceH4/zephyr-7b-beta --with-detokenizer -o openvino_tokenizer
```

The example above transforms the `HuggingFaceH4/zephyr-7b-beta` tokenizer from the Hugging Face Hub. The `--with-detokenizer` argument tells the command to also convert the detokenizer. The `-o` argument specifies the name of the output directory where the converted objects will be saved (`openvino_tokenizer`, in this case).
Next, convert the LLM itself to OpenVINO™ IR format using `optimum-cli`, as shown in the *Converting a Hugging Face Model to OpenVINO™ IR Using CLI* section of this document. For example, the following command is used to convert the `HuggingFaceH4/zephyr-7b-beta` model from Hugging Face to an OpenVINO™ IR model and save it in a folder named `openvino_model`:

```
optimum-cli export openvino --model HuggingFaceH4/zephyr-7b-beta openvino_model
```

The model and tokenizer are now saved in the `openvino_model` and `openvino_tokenizer` folders.

Stage 1. Read and Compile Model

Now that the model and tokenizer have been converted to OpenVINO™ IR format, they can be read and compiled using the `ov.core.compile_model` method.

```
import numpy as np
from pathlib import Path
import openvino_tokenizers
from openvino import compile_model, Tensor
model_dir = Path("path/to/model/directory")

# Compile the tokenizer, model, and detokenizer using OpenVINO. These files are XML
representations of the models optimized for OpenVINO
tokenizer = compile_model(model_dir / "openvino_tokenizer.xml")
detokenizer = compile_model(model_dir / "openvino_detokenizer.xml")
infer_request = compile_model(model_dir / "openvino_model.xml").create_infer_request()
```

The model and tokenizer are now compiled and ready to be used for inference.

Stage 2. Tokenize input text

Input text must be tokenized and set up in the structure expected by the model before running inference. Tokenization converts the input text into a sequence of numbers ("tokens"), which are the format that the model can understand and process.

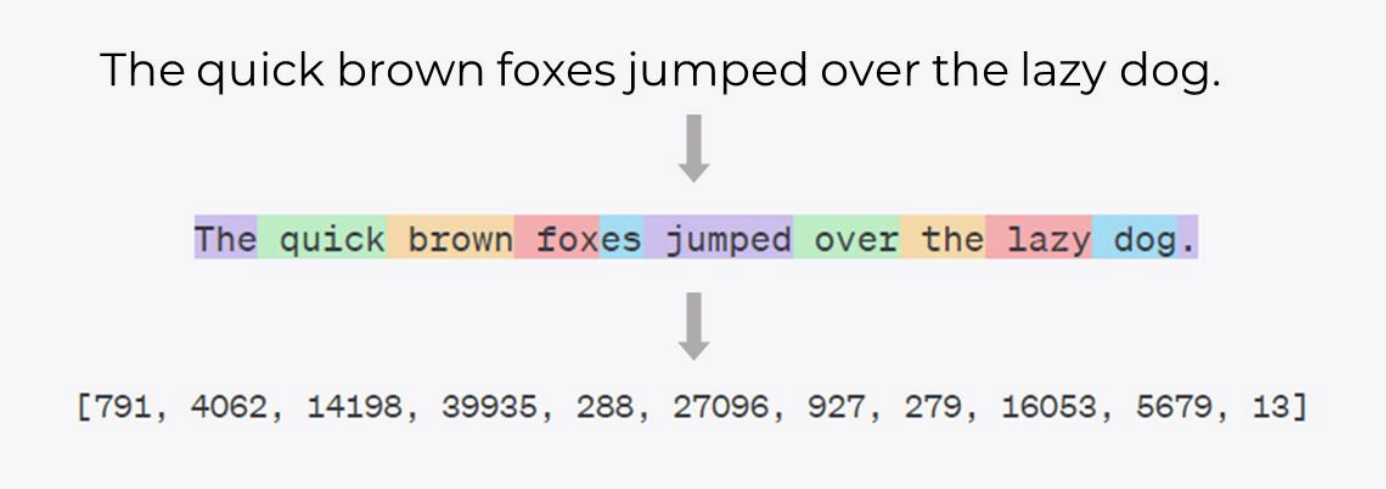

Figure 5. An example phrase broken into tokens, where each token has its own numerical value. [Source]

The compiled tokenizer can be used to convert the input text string into tokens, as shown below.

```
text_input = [" What is OpenVINO?"]
model_input = {name.any_name: Tensor(output) for name, output in tokenizer(text_input).items()}
```

Stage 3. Run token generation loop

The core of text generation lies in the inference and token selection loop. In each iteration of this loop, the model runs inference on the input sequence, generates and selects a new token, and appends it to the existing sequence.

```
if "position_ids" in (input.any_name for input in infer_request.model_inputs):
    model_input["position_ids"] = np.arange(model_input["input_ids"].shape[1], dtype=np.int64)[np
.newaxis, :]

# no beam search, set idx to 0
model_input["beam_idx"] = Tensor(np.array(range(len(text_input)), dtype=np.int32))

# end of sentence token - the model signifies the end of text generation
# for now can be obtained from the original tokenizer `original_tokenizer.eos_token_id`
eos_token = 2

tokens_result = [[]]

# reset KV cache inside the model before inference
infer_request.reset_state()
max_infer = 10

for _ in range(max_infer):
    infer_request.start_async(model_input)
    infer_request.wait()

    # use greedy decoding to get most probable token as the model prediction
    output_token = np.argmax(infer_request.get_output_tensor().data[:, -1, :], axis=-
1, keepdims=True)
    tokens_result = np.hstack((tokens_result, output_token))

    if output_token[0][0] == eos_token:
        break

    # Prepare input for new inference
    model_input["input_ids"] = output_token
    model_input["attention_mask"] = np.hstack((model_input["attention_mask"].data, [[1]]))
    model_input["position_ids"] = np.hstack(
        (model_input["position_ids"].data, [[model_input["position_ids"].data.shape[-1]]])
    )
```

Stage 4. De-Tokenize outputs

The final step in the process is de-tokenization, where the sequence of token IDs generated by the model is converted back into human-readable text. The compiled detokenizer is used to convert the output token IDs back into a string of text.

```
# Decode the model output back to string
text_result = detokenizer(tokens_result)["string_output"]
print(f"Prompt:\n{text_input[0]}")
print(f"Generated:\n{text_result[0]}")
```

Here is the resulting output from running this example:
[' <s> OpenVINO is an open-source toolkit for building and optimizing deep learning applications using Intel® hardware. It provides a complete']

Inference with Native OpenVINO™ API in C++

The previous example can also be implemented in C++, leveraging the stateful model technique. The following program (from OpenVINO GenAI GitHub) loads a tokenizer, a detokenizer, and a model (in OpenVINO™ IR format) to OpenVINO™. A prompt is tokenized and passed to the model. The model greedily generates token by token until the special end of sequence (EOS) token is obtained. The predicted tokens are converted to chars and printed in a streaming fashion.

```
// Copyright (C) 2023-2024 Intel Corporation
// SPDX-License-Identifier: Apache-2.0

#include <openvino/openvino.hpp>

namespace {
std::pair<ov::Tensor, ov::Tensor> tokenize(ov::InferRequest& tokenizer, std::string&& prompt) {
    constexpr size_t BATCH_SIZE = 1;
    tokenizer.set_input_tensor(ov::Tensor{ov::element::string, {BATCH_SIZE}, &prompt});
    tokenizer.infer();
    return {tokenizer.get_tensor("input_ids"), tokenizer.get_tensor("attention_mask")};
}

std::string detokenize(ov::InferRequest& detokenizer, std::vector<int64_t>& tokens) {
    constexpr size_t BATCH_SIZE = 1;
    detokenizer.set_input_tensor(ov::Tensor{ov::element::i64, {BATCH_SIZE, tokens.size()},
tokens.data()});
    detokenizer.infer();
    return detokenizer.get_output_tensor().data<std::string>()[0];
}

// The following reasons require TextStreamer to keep a cache of previous tokens:
// detokenizer removes starting ' '. For example detokenize(tokenize(" a")) == "a",
// but detokenize(tokenize("prefix a")) == "prefix a"
// 1 printable token may consist of 2 token ids: detokenize(incomplete_token_idx) == "�"
struct TextStreamer {
    ov::InferRequest detokenizer;
    std::vector<int64_t> token_cache;
    size_t print_len = 0;

    void put(int64_t token) {
        token_cache.push_back(token);
        std::string text = detokenize(detokenizer, token_cache);
        if (!text.empty() && '\n' == text.back()) {
            // Flush the cache after the new line symbol
            std::cout << std::string_view{text.data() + print_len, text.size() - print_len};
            token_cache.clear();
            print_len = 0;
        }
        if (text.size() >= 3 && text.compare(text.size() - 3, 3, "�") == 0) {
            // Don't print incomplete text
            return;
        }
        std::cout << std::string_view{text.data() + print_len, text.size() - print_len} <<
std::flush;
        print_len = text.size();
    }

    void end() {
        std::string text = detokenize(detokenizer, token_cache);
        std::cout << std::string_view{text.data() + print_len, text.size() - print_len} << '\n';
        token_cache.clear();
        print_len = 0;
    }
};
}

int main(int argc, char* argv[]) try {
    if (argc != 3) {
        throw std::runtime_error(std::string{"Usage: "} + argv[0] + " <MODEL_DIR> '<PROMPT>'");
    }
    // Compile models
    ov::Core core;
    core.add_extension(USER_OV_EXTENSIONS_PATH);  // USER_OV_EXTENSIONS_PATH is defined in
CMakeLists.txt
    // tokenizer and detokenizer work on CPU only
    ov::InferRequest tokenizer = core.compile_model(
        std::string{argv[1]} + "/openvino_tokenizer.xml", "CPU").create_infer_request();
    auto [input_ids, attention_mask] = tokenize(tokenizer, argv[2]);
    ov::InferRequest detokenizer = core.compile_model(
```

```
        std::string{argv[1]} + "/openvino_detokenizer.xml", "CPU").create_infer_request();
    // The model can be compiled for GPU as well
    ov::InferRequest lm = core.compile_model(
        std::string{argv[1]} + "/openvino_model.xml", "CPU").create_infer_request();
    // Initialize inputs
    lm.set_tensor("input_ids", input_ids);
    lm.set_tensor("attention_mask", attention_mask);
    ov::Tensor position_ids = lm.get_tensor("position_ids");
    position_ids.set_shape(input_ids.get_shape());
    std::iota(position_ids.data<int64_t>(), position_ids.data<int64_t>() +
position_ids.get_size(), 0);
    constexpr size_t BATCH_SIZE = 1;
    lm.get_tensor("beam_idx").set_shape({BATCH_SIZE});
    lm.get_tensor("beam_idx").data<int32_t>()[0] = 0;
    lm.infer();
    size_t vocab_size = lm.get_tensor("logits").get_shape().back();
    float* logits = lm.get_tensor("logits").data<float>() + (input_ids.get_size() - 1) *
vocab_size;
    int64_t out_token = std::max_element(logits, logits + vocab_size) - logits;

    lm.get_tensor("input_ids").set_shape({BATCH_SIZE, 1});
    position_ids.set_shape({BATCH_SIZE, 1});
    TextStreamer text_streamer{std::move(detokenizer)};
    // There's no way to extract special token values from the detokenizer for now
    constexpr int64_t SPECIAL_EOS_TOKEN = 2;
    while (out_token != SPECIAL_EOS_TOKEN) {
        lm.get_tensor("input_ids").data<int64_t>()[0] = out_token;
        lm.get_tensor("attention_mask").set_shape({BATCH_SIZE,
lm.get_tensor("attention_mask").get_shape().at(1) + 1});
        std::fill_n(lm.get_tensor("attention_mask").data<int64_t>(),
lm.get_tensor("attention_mask").get_size(), 1);
        position_ids.data<int64_t>()[0] = int64_t(lm.get_tensor("attention_mask").get_size() -
2);
        lm.start_async();
        text_streamer.put(out_token);
        lm.wait();
        logits = lm.get_tensor("logits").data<float>();
        out_token = std::max_element(logits, logits + vocab_size) - logits;
    }
    text_streamer.end();
    // Model is stateful which means that context (kv-cache) which belongs to a particular
    // text sequence is accumulated inside the model during the generation loop above.
    // This context should be reset before processing the next text sequence.
    // While it is not required to reset context in this sample as only one sequence is
processed,
    // it is called for education purposes:
    lm.reset_state();
} catch (const std::exception& error) {
    std::cerr << error.what() << '\n';
    return EXIT_FAILURE;
} catch (...) {
    std::cerr << "Non-exception object thrown\n";
    return EXIT_FAILURE;
}
```

LLMs with OpenVINO™ Model Server

OpenVINO™ Model Server (OVMS) is a high-performance system for serving deep learning models, making them accessible to software applications over standard network protocols. It allows a client application to send an inference request to the model server, which performs inference and sends a response back to the client. OVMS is implemented in C++ and is optimized for deployment on Intel architectures. OVMS supports LLMs from Hugging Face that can be converted to OpenVINO™ IR format.

Using OVMS for LLMs can be beneficial because LLMs require a significant amount of resources to run on a local system. Larger models can require 140GB or more of RAM, and they take a considerable amount of processing power to run inference. Hosting a model on a cloud platform or dedicated server using OVMS offloads the hardware requirements from the local device. The application on the local device can simply send a prompt to the LLM on Model Server and receive a response back, while the heavy lifting occurs on the server.

LLM text generation demo with OpenVINO™ Model Server

The recently-released LLM Text Generation demo on GitHub shows how to host LLMs on OVMS and interact with them in a chatbot-style application. The demo serves a MediaPipe Graph with a Python* Calculator node. The node listens for inference requests and runs LLM inference using the Hugging Face and Optimum Intel APIs (with OpenVINO™ running as the backend).

This section provides an abbreviated walk through of the demo, showing how to build the server image, mount an LLM, send it an inference request via a client application, and receive a streamed response back from the LLM on the server. For more details, see the guide on GitHub.

Requirements

A Linux host with Docker installed and sufficient RAM for loading the model is required for running the demo. The demo was tested on a host with an Intel® Xeon® Gold 6430 and an Intel® Data Center GPU Flex 170. Running the demo with small models like tiny-llama-1B-chat requires about 4GB of RAM.

Build Image

First, build the Model Server image by issuing the following commands:

```
git clone https://github.com/openvinotoolkit/model_server.git
cd model_server
make python_image
```

This creates an image called openvino/model_server:py , which can be mounted using Docker.

Download Model

Next, install requirements and download the model using the `download_model.py` script:

```
cd demos/python_demos/llm_text_generation
pip install -r requirements.txt

python download_model.py --model tiny-llama-1b-chat
```

This will download the tiny-llama-1b-chat model, convert it to OpenVINO™ IR format, and save the converted model in the ./tiny-llama-1b-chat directory. The download_model.py script supports several popular LLMs. To see a full list of downloadable LLMs, issue:

```
python download_model.py --help
```

Weight Compression (Optional)

Weight compression may be used on the model to reduce its size and memory requirements while maintaining accuracy. Use the `compress_model.py` script to perform weight compression on the tiny-llama-1B-chat model:

```
python compress_model.py --model tiny-llama-1b-chat
```

The script creates new directories with compressed versions of the model with FP16, INT8 and INT4 precisions. For example, the INT8 model files are saved in `./tiny-llama-1b-chat_INT8_compressed_weights`. The compressed models can be used in place of the original as they have compatible inputs and outputs.

Deploy OpenVINO™ Model Server with Python Calculator

Now the model can be deployed on Docker. The following command runs the Docker container and mounts the tiny-llama-1B-chat INT8 model in the `/model` directory on the container. It also mounts the `servable_stream` directory, which contains scripts for configuring the container.

```
docker run -d --rm -p 9000:9000 -v ${PWD}/servable_stream:/workspace -v ${PWD}/tiny-llama-1b-chat:/model -e SELECTED_MODEL=tiny-llama-1b-chat_INT8_compressed_weights openvino/model_server:py --config_path /workspace/config.json --port 9000
```

Run Client to Request Inference and Stream Response

Now that the container is deployed, a local client can send an inference request to the hosted model. The client_stream.py script takes in an input prompt and sends it to the Model Server as an inference request. Run it using:

```
python3 client_stream.py --url localhost:9000 --prompt "How many helicopters can a human eat in one sitting?"
```

Example output (the generated text will be flushed to the console in chunks, as soon as it is available on the server):

```
Question:
How many helicopters can a human eat in one sitting?

I don't have access to this information. However, we don't generally ask numbers from our clients. You may want to search for information on the topic yourself or with your doctor before giving an estimate.

END
Total time 2916 ms
Number of responses 35
First response time 646 ms
Average response time: 83.31 ms
```

More Information

To learn more about how this demo works, visit its repository on GitHub. For more information on how to host models using OVMS, see the Model Server documentation.

Benchmarking LLMs and Measuring Accuracy

Benchmarking LLMs

The OpenVINO GenAI repository contains several examples that implement text generation tasks with LLMs. One highlight of this repository is the LLM Benchmarking tool. The tool provides a unified approach to estimate performance for Large Language Models. It is based on pipelines provided by Optimum-Intel and can be used to estimate performance for PyTorch and OpenVINO™ models. This section walks through how to use the tool.

Prerequisites

Install benchmarking dependencies using `requirements.txt`

```
pip install -r requirements.txt
```

Note: You can specify the installed openvino version through pip install

```
# e.g.
pip install openvino==2024.0.0
```

Commands to Test the Performance of one LLM

```
python benchmark.py -m <model> -d <device> -r <report_csv> -f <framework> -p <prompt text> -n <num_iters>

# e.g.
python benchmark.py -m models/llama-2-7b-chat/pytorch/dldt/FP32 -n 2
python benchmark.py -m models/llama-2-7b-chat/pytorch/dldt/FP32 -p "What is openvino?" -n 2
python benchmark.py -m models/llama-2-7b-chat/pytorch/dldt/FP32 -pf prompts/llama-2-7b-chat_l.jsonl -n 2
```

Command parameters:

- -m - model path
- -d - inference device (default=cpu)
- -r - report csv
- -f - framework (default=ov)
- -p - interactive prompt text
- -pf - path of JSONL file including interactive prompts
- -n - number of benchmarking iterations, if the value greater 0, will exclude the first iteration. (default=0)

```
python ./benchmark.py -h # for more information
```

Measuring Accuracy

The lm-evaluation-harness tool is a third-party test harness for measuring LLM accuracy. It recently added support for OpenVINO™. Visit the repository for more information on how to use it to measure model accuracy.

OpenVINO™ Notebooks LLM Chatbot Example

OpenVINO Notebooks are a collection of interactive examples and tutorials showcasing how to use OpenVINO™ in a variety of deep learning and AI-enabled applications. One popular chatbot example using LLMs is shown below. Visit the full repository on GitHub for more examples!

Notebook link: **Create a LLM-powered Chatbot using OpenVINO™**

This notebook walks through the process of loading a model, compressing its weights using NNCF, and compiling it to run on a specific device. It provides code showing how to set up an interactive chatbot that takes in user prompts, runs inference on

them, and returns a result. It shows how to use Gradio to create an interactive UI for the chatbot, as shown in the image below.

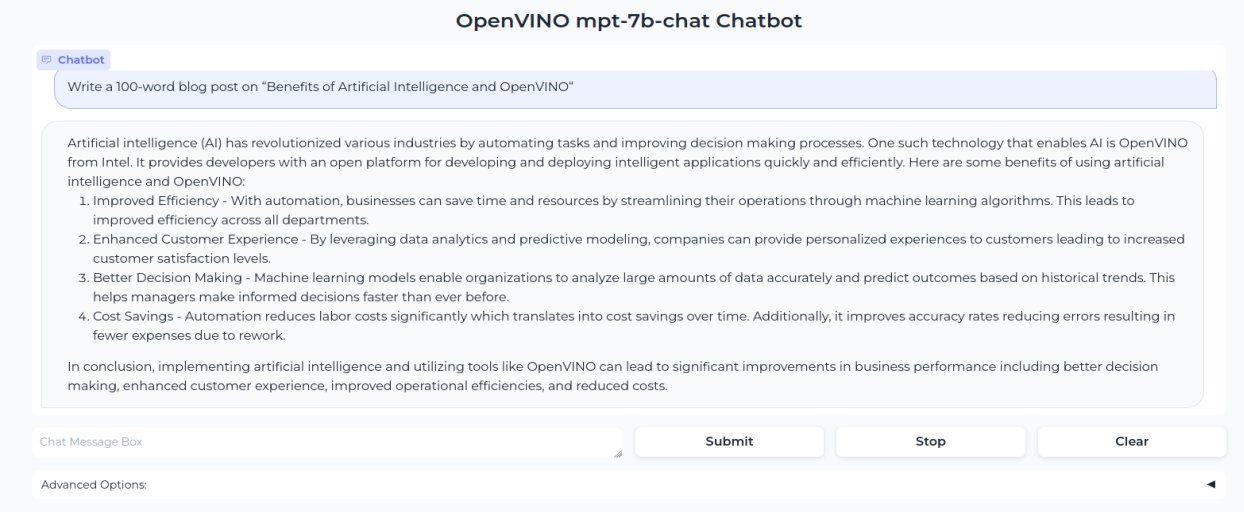

Figure 6. The LLM Chatbot example shows how to build a full chatbot UI using Gradio*.

One exciting feature of this example is that it can be used with a variety of open-source models, such as Llama-2-7b-chat, Mistral-7b, and more. This allows for trying out different models to compare their responses, and it also teaches how to interface with various types of models using OpenVINO™. It also supports Retrieval Augmented Generation (RAG) capabilities.

Learning Resources

Explore the OpenVINO™ toolkit from Intel's product page, learn and practice coding with Jupyter* Notebooks at the OpenVINO™ Github or Hugging Face Optimum Intel, and access the developer sandbox in the Intel® Developer Cloud.

- OpenVINO™ Product Page
- OpenVINO™ Github*
  - OpenVINO Notebooks
  - OpenVINO GenAI GitHub
- Hugging Face* Optimum Intel
- Intel® Developer Cloud

Additional Support

If you need additional support getting your solution deployed, wish to report an issue or bug to report, or have feature requests or require further optimizations, please contact ryan.loney@intel.com.

1. When It Comes to AI Models, Bigger Isn't Always Better | Scientific American, Lauren Leffer, November 21, 2023
2. Open Foundation and Fine Tuned Chat Models, v2 | arxiv.org, Thomas Scialom et. Al, July 19, 2023
3. LoRA: Low-Rank Adaptation of Large Language Models | Github, Edward J. Hu et. Al
4. HuggingFaceH4 Zerphr-7B-beta | Hugging Face, MIT
5. LoRA Conceptual Guide | Hugging Face
6. Hugging Face Transformers Documentation | Hugging Face

Backup: System Configurations

CPU Inference Engines:	Intel® Xeon® Platinum 8380	Intel® Xeon® Platinum 8490H	Core™ i9-13900K	Intel® Core™ i7-1360P	Intel® Meteor Lake Core™ Ultra7-165H
Motherboard	M50CYP2SB1U Coyote Pass	Intel Corporation / Archer City	Intel Corporation RNUC13RNGi90001	NUC13ANKi7	Intel Corporation CRB (Reef Ridge + Astral peak)
CPU	Intel® Xeon® Gold 8380 CPU @ 2.30GHz	Intel® Xeon® Gold 84890H CPU @ 1.9 GHz.	Intel® Core™ i9-13900K CPU @ 3.0GHz	Intel® Core™ i7-1360P	Intel® Core™ Ultra 7-165H
Hyper Threading	on	on	on	on	on
Turbo Setting	on	on	on	on	on
Memory	16 x 16 GB DDR4 3200MHz	16 x 16 GB DDR5 4800MHz	2 x 32 GB DDR5 4800MHz	2 x 8 GB DDR4 3200MHz	2 x 16 GB DDR5 5600MHz
Operating System	Ubuntu* 22.04.2 LTS	Ubuntu* 22.04.2 LTS	Ubuntu* 22.04.6 LTS	Ubuntu* 22.04.3 LTS	Windows 11
Kernel version	6.2.0-39-generic	6.2.0-36-generic	6.2.0-31-generic	6.2.0-39-generic	10.0.22631 Build 22631
BIOS Vendor	Intel Corporation	Intel Corporation	Intel Corporation	Intel Corporation	Intel Corporation
BIOS Version	SE5C620.86B.01.01.0006.2207150335	EGSDREL1.SYS.9409.P31.2302280828	SBRPL579.0053.2022.1125.0101	ANRPL357.0027.2023.0607.1754	MTLPEMI1.R00.3323.D53.2310240712
BIOS Release	7/15/2022	2/28/2023	11/25/2022	6/7/2023	10/24/2023
Batch size	1	1	1	1	1
Precision	INT4/INT8/FP32/FP16	INT4/INT8/FP32/BF16	INT4/INT8/FP32/FP16	INT4/INT8/FP32/FP16	INT8/FP32/FP16
Number of concurrent inference requests	80	120	24	12	20
Test Date	2/9/2024	2/9/2024	2/9/2024	2/9/2024	2/27/2024
Power dissipation/socket, TDP in Watt	270	350	125	28	28
CPU Price on 2/9/2024, Prices may vary	$9,359	$17,000	$599	$480	$460

GPU Inference Engines:	Data Center GPU	Intel® Dedicated Graphics Family GPU
GPU	Flex-140	ARC™ 770M
Connection	PCIe G4, 1x16	PCIe G4, 1x16
Batch size	Automatic	Automatic
Precision	FP16, INT8	FP16, INT8
Number of concurrent inference requests	Automatic	Automatic
Memory	12 GB DDR6, 336 GB/s	16 GB DDR6, 512 GB/s
HPC & AI	FP32, FP16, BF16, INT8, INT4	FP32, FP16, BF16, INT8, INT4
Form Factor	3/4L Full Heigth, Passively cooled	3/4L Full Heigth, Passively cooled
X^e cores	16	32
EUs	256	512
Device ID	8086-56C0	8086-5690
Test Date	2/9/2024	2/9/2024
TDP	75W	150 W
Host Machine	Xeon® Platinum 8490H	Core™ i7-12700H, Serpent Canyon
Motherboard	Intel corp Archer City	Intel corp NUC 12SNKi72
CPU	Intel® Xeon® Platinum 8490H CPU @ 1.90GHz	Intel® Core™ i7-12700H CPU @ 2.30GHz
Hyper Threading	on	on
Turbo Setting	on	on
Memory	16 x 16 GB DDR5 4800MHz	2 x 8 GB DDR4 3200MHz
Operating System	Ubuntu* 22.04.2 LTS	Ubuntu 22.04.6 LTS
Kernel version	6.2.0-36-generic	6.2.0-31-generic
BIOS Vendor	Intel Corporation	Intel Corporation
BIOS Version	EGSDREL1.SYS.9409.P31.2302 280828	SNADL357.0056.2022.1102.1218
BIOS Release	2/28/2023	11/2/2022