Technical Paper

Health, Education and Consumer Industries

Optimizing Large Language Model Inference on Intel® Processors with IPEX and IPEX-LLM

Optimized Inference Strategies for LLMs on Intel® Processors: Exploring IPEX and IPEX-LLM Solutions

Authors

Syafie Nizam
Solution Architect

Kuan Heng Lee
Solution Architect

Mee Sim Lai
Solution Architect

Yueh Shen Soon
Platform Solution Architect

Abstract

This whitepaper explores the deployment of Large Language Models (LLMs) on CPU using the Intel® Extension for PyTorch (IPEX) and the Intel® LLM Library for PyTorch (IPEX-LLM). It delves into various hardware and software architectures to optimize the performance, resource utilization, and response times of LLMs in real-world applications. The paper discusses two inferencing approaches: batch inferencing and multi-instance inferencing, providing insights into the suitability of these approaches for different scenarios, such as real-time applications and high-throughput tasks. Additionally, it examines the necessary hardware configurations to support these models, including detailed system specifications and software setups. This document is specifically designed for developers who are interested in performing inference on Intel® Processors and understanding the performance capabilities when running on such processors. It aims to facilitate developers in evaluating Intel® Processors by testing out performance through practical benchmarks. The paper also includes a quick start guide for developers to set up the environment, install dependencies, and execute inference benchmarks to evaluate performance effectively.

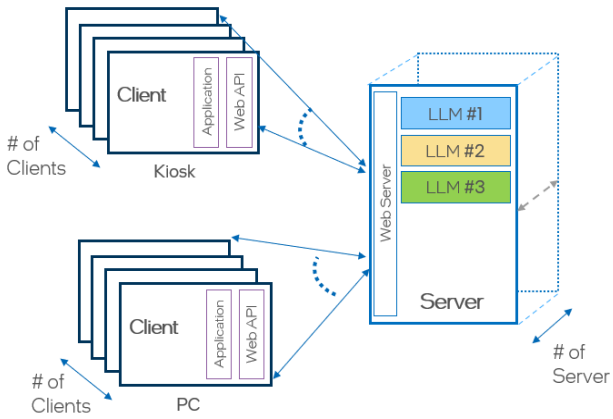
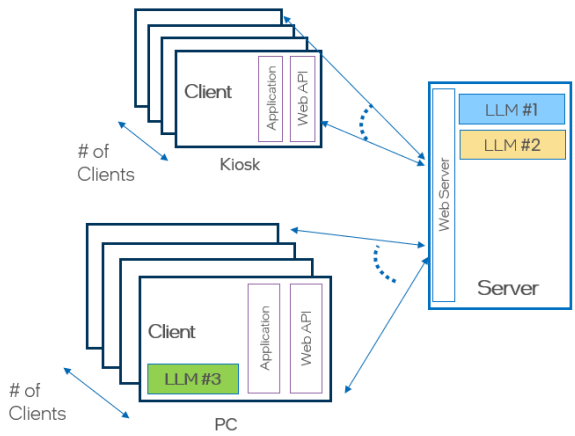
Introduction

The deployment of Large Language Models (LLMs) in various application scenarios requires careful consideration of the system architecture to optimize performance, resource utilization, and response times. This paper explores different hardware architectures for hosting the LLM models, including different LLM inferencing approaches such as batch inferencing and multi-instance inferencing. Software implementation guidance is also provided for developers to optimize their appropriate usage. This document is specifically crafted for developers aiming to perform inference on Intel Processors and to understand and evaluate the performance capabilities of these processors. It includes practical insights for developers to set up the environment, install dependencies, and execute inference benchmarks to assess performance comprehensively while offering a quick start guide to facilitate these processes.

Hardware Architecture

Building and deploying Large Language Models (LLMs) presents several significant hardware challenges, including the need for immense computational power, vast memory and data storage, and efficient energy consumption. Scaling the infrastructure to manage the massive parallel processing, high-speed networking and specialized hardware required for inferencing is crucial, while ensuring system reliability and fault tolerance is also critical. Additionally, the cost of acquiring, maintaining, and operating this sophisticated hardware is substantial, highlighting the complexity of supporting advanced AI systems.

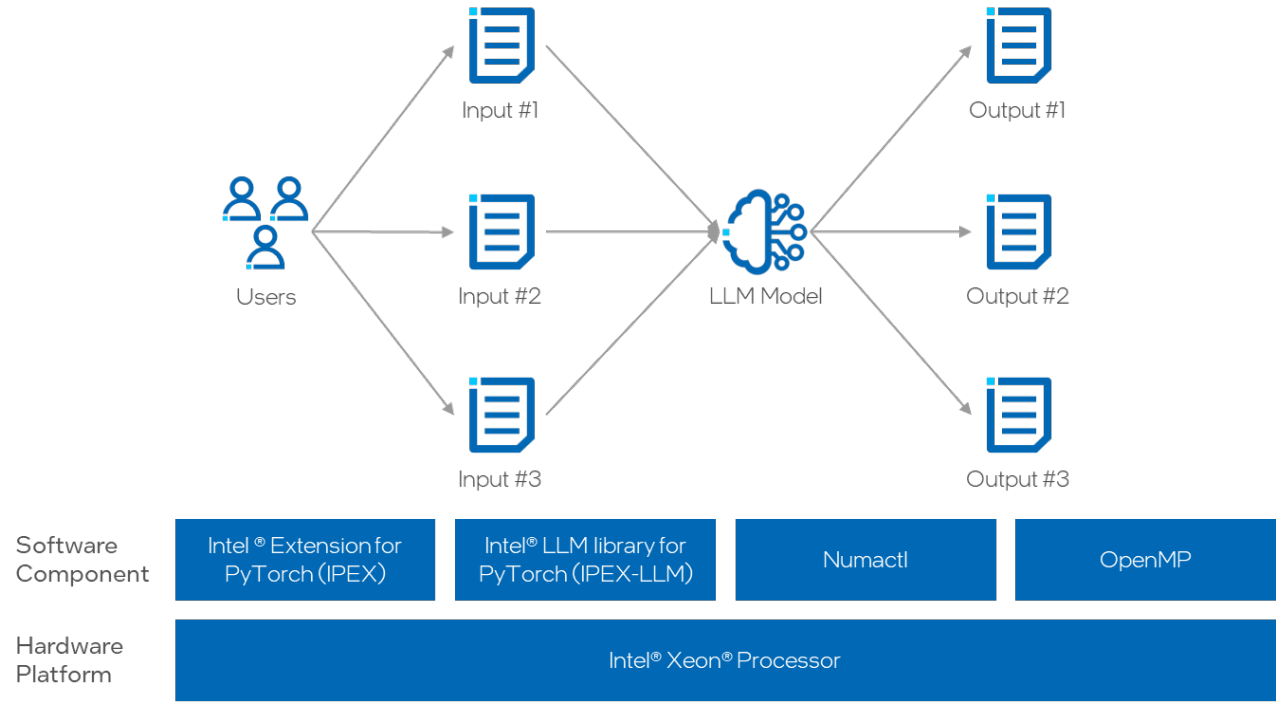
The following table summarizes some of the considerations of hosting LLM with the following two different approaches:

Architecture	**Consolidated**	**Distributed**
Diagram		
Setup	Hosting multiple LLMs in a scalable Server.	Distributed LLMs between Client PCs and Server.
Pros	Easy software maintenance. For example, model updates.	Improved latency as the LLMs are hosted on the Client PCs, allowing them to process requests locally.
Cons	Expecting increased latency due to communication between Client PCs and Server for LLM requests.	Managing software maintenance, such as model updates, across multiple Client PCs can be challenging.

Software Architecture

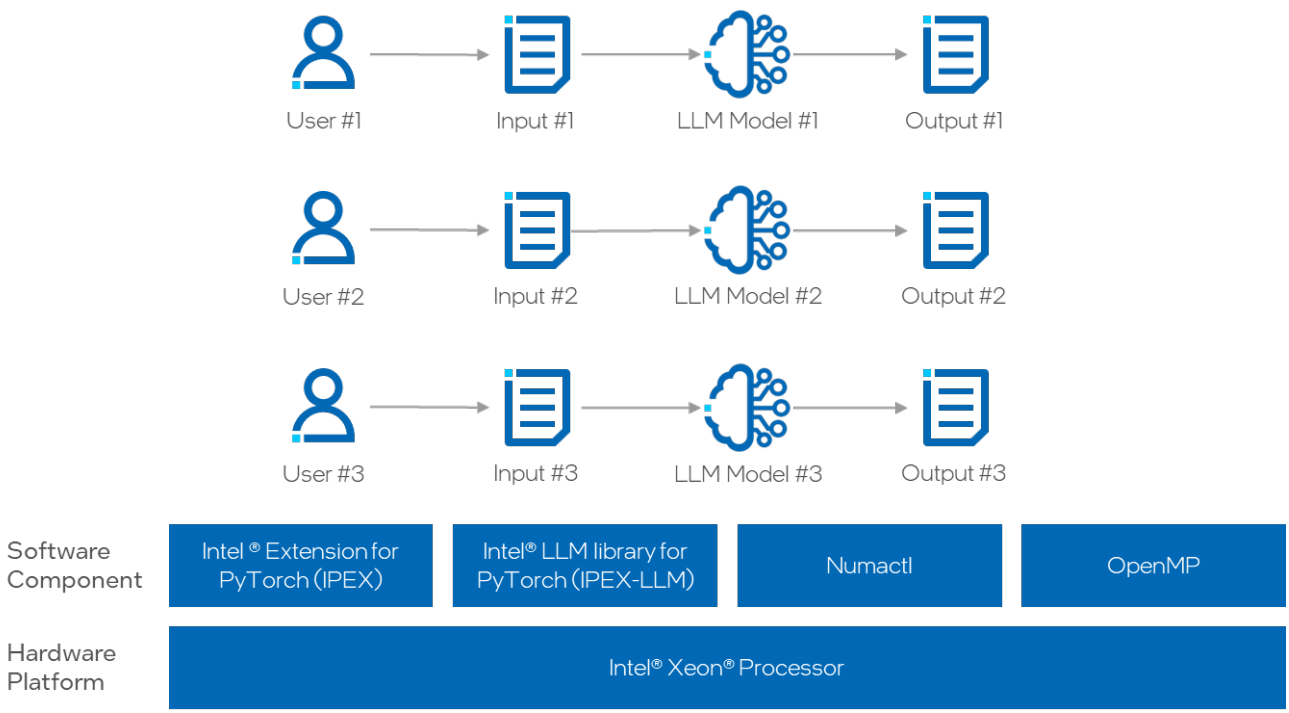
Batch Inferencing

The batch inferencing approach involves deploying a single LLM instance on a system by handling multiple input requests simultaneously by processing them in batches. This method is particularly effective for applications requiring high throughput, such as document summarization tasks and offline analytics, where multiple inputs can be grouped and processed together. However, the requirement to accumulate inputs into batches can introduce latency, making this approach less suitable for real-time applications where prompt responses are critical.



Multi-instance Inferencing

The multi-instance inference approach involves deploying multiple LLM instances on a system, where each instance processes its own independently. This setup is suitable for handling different types of LLM models on the same system that requires low latency and high concurrency. For example, the system needs to handle concurrent requests for multiple LLMs, such as conversational chatbots and video summarization, on a single system. By allowing each request to be processed immediately by a dedicated model instance, this approach minimizes response times and makes it ideal for real-time applications. Furthermore, the ability to scale by adding more model instances enhances the system's capacity to handle high volumes of concurrent requests. However, if the system lacks the necessary computational power or memory to efficiently support concurrent requests, it can lead to resource contention and system instability.



The following table summarizes the differences between the two inferencing approaches:

Aspect	Batch Inferencing	Multi-Instance Inferencing
Setup	Single LLM instance per system.	Multiple LLM instances per system.
Request Handling	Multiple requests are processed in batches.	Each request is processed by an independent model instance.
Latency	Higher latency due to batch accumulation.	Low latency, immediate processing of each request.
Throughput	High throughput; efficient resource utilization for batch processing.	High concurrency, scales with more mode instances.
Usage	Offline applications with high throughput applications.	Real-time applications with high-concurrency environments.

Software Components

The software components used to run LLM model inference are the Intel® Extension for PyTorch (IPEX), Intel® LLM Library for PyTorch (IPEX-LLM), and Numactl. IPEX is an extension of native PyTorch, incorporating the latest optimization features to enhance performance on Intel hardware. These enhancements utilize Intel® AVX-512 VNNI and Intel® AMX on Intel Processors and Intel® XMX AI engines on Intel discrete GPUs.

IPEX-LLM is a unique PyTorch library specifically designed for running LLM on Intel® Processors and GPUs. It introduces advanced features that supports on-the-fly optimization model conversion, significantly enhancing the experience for developers. This powerful capability extends the potential of running LLMs beyond traditional GPUs using the HuggingFace Transformers and can be expanded to Intel integrated GPUs.

On the other hand, Numactl is a Linux tool that allows user control of NUMA policy for processes or shared memory. This can optimize performance by managing memory and CPU allocation for workloads. Additionally, OpenMP is an implementation of multithreading, enabling workloads to run on multiple cores or processors simultaneously to improve their performance.

Quick Start

Create Environment

To get started with IPEX-LLM inferencing, set up a Python environment.

```
conda create -n llm python=3.11
conda activate llm
```

Install Dependencies

Once the environment is created and activated, install the necessary dependencies. It is important to note that IPEX-LLM and IPEX are two different libraries. As of 19/8/2024, IPEX-LLM only supports Intel® Extension for PyTorch version 2.2.0.

```
pip install intel-extension-for-pytorch==2.2.0
pip install oneccl_bind_pt==2.2.0 --extra-index-url https://pytorch-extension.intel.com/release-whl/stable/cpu/us/
pip install --pre --upgrade ipex-llm[all] --extra-index-url https://download.pytorch.org/whl/cpu
pip install torch==2.2.0 torchvision==0.17.0 torchaudio==2.2.0 --index-url https://download.pytorch.org/whl/cpu
pip install transformers==4.36.2
pip install omegaconf
pip install pandas
```

Clone IPEX-LLM repo

```
git clone https://github.com/intel-analytics/ipex-llm.git
cd ipex-llm/python/llm/dev/benchmark/all-in-one/
```

Check CPU Information

Understanding the CPU architecture and capabilities is crucial before performing any inferencing workload or benchmark. This information helps in optimizing the benchmarking process and ensures that the system's resources are utilized efficiently. To gather essential CPU details, use the *lscpu* command on Linux. This command provides information such as the number of cores, sockets, and how CPU cores are distributed across the machine. This will be particularly useful later when employing the *numactl* tool for CPU core binding and memory binding. For instance, executing *lscpu* on a system with two Intel® Xeon® Gold 6448Y CPUs would yield the following details:

```
$ lscpu
...
CPU(s):              128
On-line CPU(s) list: 0-127
Thread(s) per core:  2
Core(s) per socket:  32
Socket(s):           2
NUMA node(s):        2
...
Model name:          Intel(R) Xeon(R) Gold 6448Y
...
NUMA node0 CPU(s):   0-31,64-95
NUMA node1 CPU(s):   32-63,96-127
...
```

When indexing CPU cores, physical cores are typically listed before logical cores. In this scenario, the first 32 cores (0-31) are physical cores on the first socket (node 0), and the next 32 cores (32-63) are physical cores on the second socket (node 1). Logical cores follow this sequence: cores 64-95 are the 32 logical cores on the first socket (node 0), and cores 96-127 are the 32 logical cores on the second socket (node 1). For optimal performance, we recommend avoiding the use of logical cores when running Intel® Extension for PyTorch.

Batch Inferencing

Configure YAML File

Modify the *config.yaml* file in this folder/directory for your benchmark configuration.

```yaml
repo_id:
  - 'meta-llama/Meta-Llama-3-8B'
local_model_hub: 'local-model-path'
warm_up: 1
num_trials: 3
num_beams: 1
low_bit: 'sym_int4'
batch_size: 1
in_out_pairs:
  - '128-32'
test_api:
  - "bigdl_ipex_int4"
cpu_embedding: False
streaming: False
use_fp16_torch_dtype: True
task: 'continuation'
```

The following are the details of the parameters in *config.yaml*:

- `repo_id:` The model to be used. Please specify the actual model's name and its organization from HuggingFace. For example, 'meta-llama/Meta-Llama-3-8B. Suggested to test on a different model to see different model's performance.
- `local_model_hub:` If the model has been downloaded to the local storage, paste its local path here, for example, 'home/user/models'.
- `batch_size:` The number of samples on which the models make predictions in one forward pass. Change this value to greater than 1 to support batch inferencing.
- `in_out_pairs:` Number of pairs of input and output tokens.
- `test_api:` Different test functions for different machines. Use `bigdl_ipex_int4` as this is the optimized version of transformers for Intel® Processors. Try different precisions such as `bigdl_ipex_int8` and `bigdl_ipex_bf16`, we highly encourage the user to lower the precision for better throughput.

Configure bash script (run-spr.sh)

```
#!/bin/bash
source ipex-llm-init -t
export OMP_NUM_THREADS=32

numactl -C 0-31 -m 0 python $(dirname "$0")/run.py
```

The value of `OMP_NUM_THREADS` should match the number of CPU cores specified by `numactl -C`. For example, on a system with the Intel® Xeon® 6448Y processor with 32 cores, you would set `OMP_NUM_THREADS=32` and `numactl -C 0-31`. This bash script will execute the inferencing benchmark workload on the physical cores (0-31) of the first socket (node 0). To demonstrate the versatility of socket usage, consider this scenario: one socket, such as the one mentioned, can be dedicated to running LLMs for intensive computational tasks, while the other socket can be reserved for general-purpose compute tasks such as applications, media processing, and other activities. This strategic allocation of resources ensures that performance-critical tasks can operate without interference, optimizing both throughput and efficiency.

Run Benchmark

The benchmark script is designed to evaluate the system's performance by measuring various metrics that reflect the system's efficiency in handling computational tasks. It helps users determine the operational capacity and identify performance bottlenecks. The expected usability throughput or latency should align with the user's requirements for optimal performance, making it easy to interpret whether the numerical results are within a desirable range.

```
$ bash run-spr.sh
```

The benchmark result will be generated in a CSV file. The content of the file will be the name of the model, 1st token latency (ms), Nth token latency (ms/token), model loading time (s), and other relevant benchmark metrics. This generated data can be used to validate whether the actual input/output tokens are consistent with the specific parameters in the *config.yaml* file. Additionally, this data can be used to determine if the specific LLM models meet the use case-specific requirements, such as throughput or latency.

Multi-Instance Inferencing Benchmark

We can utilize *numactl* and *bash script* to simulate a multi-instance inferencing.

Configure bash script (run-spr.sh)

```
#!/bin/bash
source ipex-llm-init -t
export OMP_NUM_THREADS=16

# 16(2)
numactl -C 0-15 -m 0 python $(dirname "$0")/run.py &
numactl -C 16-31 -m 0 python $(dirname "$0")/run.py
```

The tested system has two sockets and 32 cores on each socket. We can experiment with running multi-instance inferencing by binding each workload to specific cores. For example, this bash script runs two inference workloads: the first workload utilizes cores 0-15 and the second workload utilizes cores 16-31. The memory bind `-m` option specifies the memory policy for the process, -m 0 indicates that the process should use the memory associated with the socket 0. The environment variable `OMP_NUM_THREADS` sets the number of threads used for parallel regions. This can be used along with the *numactl* settings, as shown in the example above, the command runs on cores 0-15 and 16-31 with 16 OpenMP threads each.

```
#!/bin/bash
source ipex-llm-init -t
export OMP_NUM_THREADS=16

# 16(4)
numactl -C 0-15 -m 0 python $(dirname "$0")/run.py &
numactl -C 16-31 -m 0 python $(dirname "$0")/run.py &
numactl -C 32-47 -m 1 python $(dirname "$0")/run.py &
numactl -C 48-63 -m 1 python $(dirname "$0")/run.py
```

The bash script above provides another example of running four inference workloads on both sockets. The first and second workloads are assigned to cores 0-15 and 16-31, respectively, which are both situated on socket 0 of the first socket. On the other hand, the third and fourth workloads are assigned to cores 32-47 and 48-63, respectively, which are located on socket 1 of the second socket. It is important to note that the memory bind, denoted as -m, corresponds to the socket where the workload is being processed. This is due to high-speed cache being present within a single socket; hence it is a good idea to avoid computations that cross over different sockets. From the perspective of memory access, it is significantly faster to bind memory access locally as opposed to accessing remote memories[1].

Conclusion

The deployment of LLMs on Intel® Processor platforms, specifically using IPEX and IPEX-LLM, presents a viable and efficient alternative to LLM hosting using GPUs. The exploration of batch and multi-instance inferencing reveals distinct advantages in terms of throughput and response times, making them suitable for different application requirements. The hardware and software configurations detailed in this paper provide a robust foundation for implementing and scaling LLM solutions. By following the provided guidelines for benchmarks, developers can optimize LLM deployments to meet specific performance criteria, ensuring efficient utilization of resources and optimal operational performance. This technical paper serves as a comprehensive guide for developers looking to harness the power of LLMs on CPU architectures, paving the way for innovative applications in various industries.

System Configuration

Hardware Configurations	
System	Quanta Cloud Technology Inc. QuantaGrid D54Q-2U
CPU Model	Intel® Xeon® Gold 6448Y
Microarchitecture	SPR_MCC
Sockets	2
Cores per Socket	32
Hyperthreading	Enabled
CPUs	128
Software Configurations	
Workload	Meta LLAMA-3-8B Inferencing
Application	ipex-llm==2.1.0b20240819, intel-extension-for-pytorch==2.2.0
Tools/Compilers	Python
Middleware, Framework, Runtimes	transformers==4.36.0, tokenizers==0.15.2, torch==2.2.0+cpu, torchaudio==2.2.0+cpu, torchvision==0.17.0+cpu, oneccl_bind_pt==2.2.0, omegaconf, pandas

1. Performance Tuning Guide — Intel Extension for PyTorch

Document number: 834133 | Revision 1.0 | September 2024