CS6886W - System Engineering for DL
Assignment-2: Performance Analysis of GPT-2

Computer Science and Engineering Department
IIT Madras, Chennai, Tamil Nadu

Due Date: Thursday, November 13, 2025

1 Objective

This assignment focuses on benchmarking the `GPT-2 medium` model using the open-source `llama.cpp` framework under multiple configurations. The goal is to evaluate performance using systematic experiments and analyze results through the Roofline performance model.

Learning Outcomes

By the end of this assignment, students will be able to:

1. Work with an open-source Large Language Model (`GPT-2 medium`) using `llama.cpp`.
2. Understand the impact of compilation flags when running the model (**Naive Execution**).
3. Evaluate the effect of hardware-specific vendor libraries (**Near Optimal Execution**).
4. Analyze the impact of parallelism on LLM execution (**Fully Optimal Execution**).

2 Tasks

Task 1: Install llama.cpp from Source(5 Marks)

Steps to setup llama.cpp

Click this **link** for Official Build instructions

Deliverables:

- List down the steps followed to set-up and install `llama.cpp`.
- Submit a screenshot showing successful build/installation.

Task 2: Setting up GPT Model(5 Marks)

A. Download GPT-2 Medium. Clone the model repository directly from Hugging Face:

Download model

```
git clone https://huggingface.co/openai-community/gpt2-medium
```

B. Convert to `.gguf` : Use the official conversion script in `llama.cpp`

Convert HF model to GGUF

```
python3 convert_hf_to_gguf.py <model_path> --outfile <out.gguf>
```

C. Run a Sanity Benchmark. Benchmark the converted model using `llama-bench`:

Quick sanity check

```
./llama-bench -m gpt2-medium.gguf -p 0 -n 256
```

Tip: Use `./llama-bench --help` to explore additional flags.

Deliverable: Submit screenshots after completing each step (download, conversion, and benchmark run).

Task 3: Naive Execution (No Parallelism)(10 Marks)

In this task, you will run the model without any form of data-level, instruction-level, or thread-level parallelism. To achieve this, you need to re-build `llama.cpp` by disabling all acceleration flags.

Rebuild llama.cpp without parallelism

```
cmake -B build \
  -DGGML_CPU_GENERIC=ON \
  -DGGML_NATIVE=OFF \
  -DGGML_AVX=OFF \
  -DGGML_AVX2=OFF \
  -DGGML_AVX512=OFF \
  -DGGML_SSE42=OFF \
  -DGGML_F16C=OFF \
  -DGGML_FMA=OFF
```

Run Benchmark: After compilation, run the benchmark in single-thread mode:

Benchmark without parallelism

```
./llama-bench -m gpt2-medium.gguf -p 0 -n 256 -t 1
```

Note: This setup forms the baseline performance (**naive execution**).

Deliverable:

- Submit benchmark results from the single-thread run.

Task 4: Default Execution(5 Marks)

In this task, you will build `llama.cpp` using the official default build configuration. This build enables instruction-level parallelism (SIMD/vectorization) and other CPU-specific optimizations by default. However, for consistency with Task 3, you will still run the benchmark with only a single thread.

A. Build with default settings

Default CPU build

```
cmake -B build
cmake --build build --config Release
```

B. Run Benchmark

Benchmark with single thread

```
./llama-bench -m gpt2-medium.gguf -p 0 -n 256 -t 1
```

Deliverables:

- Submit a screenshot of the benchmark output for the single-threaded execution.

Task 5: Near-Optimal Execution with Intel MKL(10 Marks)

In this task, you will rebuild `llama.cpp` by enabling BLAS support with Intel MKL, which provides near-optimal performance through highly tuned vendor libraries. This step requires setting up Intel MKL libraries on your system. Please refer to the official build instructions for details on Intel MKL setup.

A. Rebuild with Intel MKL. Use the following `cmake` command to build `llama.cpp` with BLAS and Intel MKL support:

Build with Intel MKL (Near Optimal)

```
cmake -B build \
  -DGGML_BLAS=ON \
  -DGGML_BLAS_VENDOR=Intel10_64lp \
  -DCMAKE_C_COMPILER=icx \
  -DCMAKE_CXX_COMPILER=icpx \
  -DGGML_NATIVE=ON
```

B. Run Benchmark Once built, run the benchmark with a single thread:

Single-threaded Benchmark with MKL

```
./llama-bench -m gpt2-medium.gguf -p 0 -n 256 -t 1
```

Deliverables:

- Submit a screenshot of the benchmark output for the single-threaded execution.

Task 6: Report Floating-Point Performance Counters(5 Marks)

In this task, you will collect and report all available floating-point related performance counters using the `perf` tool. This step will help understand the micro-architectural features of your CPU when running the LLM benchmark.

Steps.

- List down all the relevant floating point and memory traffic performance counters available on your machine for computing Operation Intensity.
- Examples of counters to report (exact names may vary by CPU):
  - **Floating Point Counters:**
    - `fp_arith_inst_retired.scalar_single`
    - `fp_arith_inst_retired.scalar_double`
    - `fp_arith_inst_retired.128b_packed_single`
    - `fp_arith_inst_retired.256b_packed_single`
    - `fp_arith_inst_retired.128b_packed_double`
    - `fp_arith_inst_retired.256b_packed_double`
    - `fp_arith_inst_retired.vector`
  - **Memory Counter:**
    - `uncore_imc_free_running/data_total`

Deliverables.

- Tabulate the counters with a short description of what each counter measures.

Task 7: Performance Counters and Roofline Analysis(30 Marks)

In this task, you will collect performance counters for all build variants of `llama.cpp` (Tasks 3, 4, and 5) and use them to derive Operation Intensity (OI) and perform a Roofline analysis.

Deliverables.

- Submit the performance counter values collected using `perf` for each variant.
- Provide the Operation Intensity derivation for each case.
- Submit peak memory bandwidth and peak compute capacity values of your system.
- Submit the Roofline analysis plot overlaying all three variants.
- Provide a short commentary on where each variant sits (memory-bound vs compute-bound).

Task 8: Fully Optimal Execution with Thread Scaling(30 Marks)

In this task, we fix the Intel MKL-enabled build (from Task 5) and only vary the number of threads at runtime. This experiment helps evaluate how well the optimized build scales with available CPU parallelism.

Steps.

- Run the benchmark multiple times with varying thread counts (1, 2, 4, 8, 12, 16, 20, 24, 28, 32)
- Collect performance counters for each run using `perf`
- Derive Operation Intensity for each thread configuration
- Extend the Roofline plot to include the scaling results

Deliverables.

- Submit benchmark logs and performance counter outputs for all tested thread counts.
- Provide a Roofline plot showing scaling behavior across thread counts.
- Add a brief discussion on how close the MKL build approaches peak compute throughput as threads increase.