CS6886W - System Engineering for Deep Learning

Assignment 2: Performance Analysis of GPT-2

Name: **Akshay Rajesh** Roll Number: **CS24M504**

Date: 19/11/2025

Objective

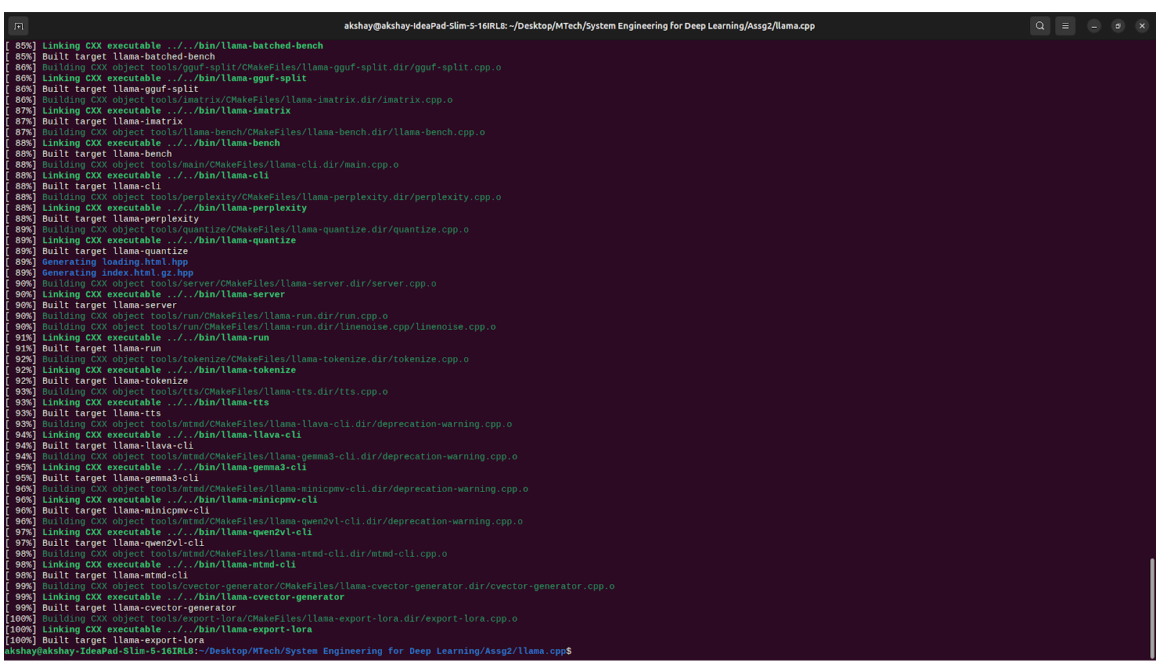
This assignment focuses on benchmarking the GPT-2 medium model using the open-source llama.cpp framework under multiple configurations and analyzing the performance using the Roofline model.

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

List the steps followed to set up and install llama.cpp.

- `sudo apt update`
- `sudo apt install cmake`
- `cmake -B build`
- `cmake --build build --config Release`

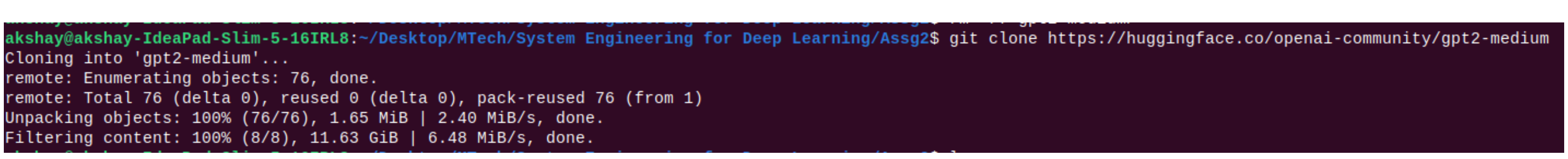
Submit a screenshot showing successful build/installation.



Task 2: Setting up GPT Model (5 Marks)

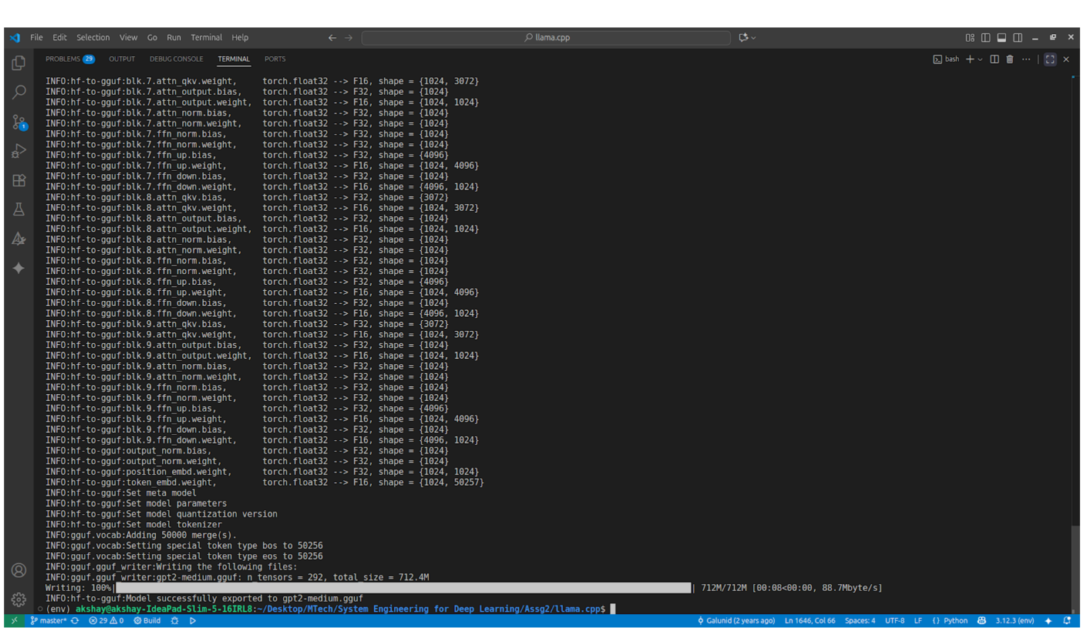
A. Download GPT-2 Medium.

- `curl -s https://packagecloud.io/install/repositories/github/git-lfs/script.deb.sh | sudo bash`
- `sudo apt-get install git-lfs`
- `git clone https://huggingface.co/openai-community/gpt2-medium`

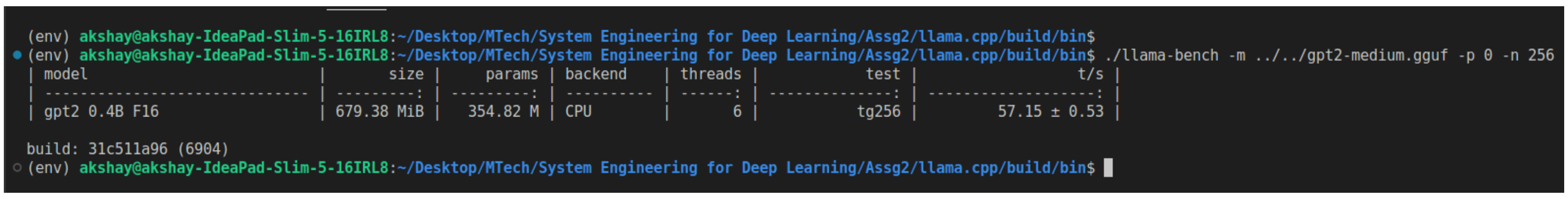


B. Convert to .gguf

- `python3 convert_hf_to_gguf.py ../gpt2-medium/ --outfile gpt2-medium.gguf`



C. Run a sanity benchmark

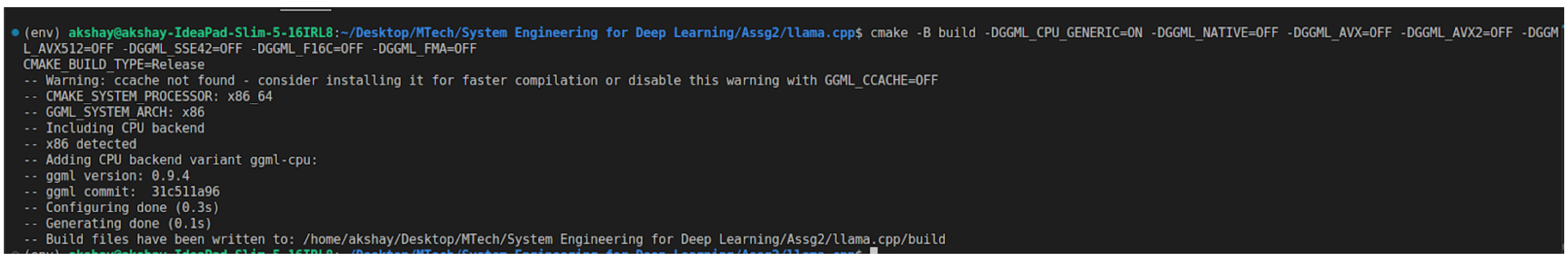


```
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp/build/bin$
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp/build/bin$ ./llama-bench -m ../../gpt2-medium.gguf -p 0 -n 256
| model                          |       size |     params | backend    | threads |            test |                  t/s |
| ------------------------------ | ---------: | ---------: | ---------- | ------: | --------------: | -------------------: |
| gpt2 0.4B F16                  | 679.38 MiB |   354.82 M | CPU        |       6 |           tg256 |         57.15 ± 0.53 |

build: 31c511a96 (6904)
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp/build/bin$
```

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

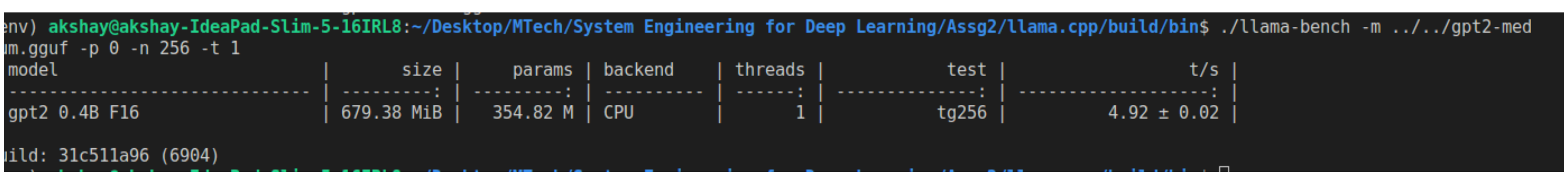
A. Steps to rebuild llama.cpp without parallelism.

- `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`
- `cmake –build build –config Release`



```
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp$ cmake -B build -DGGML_CPU_GENERIC=ON -DGGML_NATIVE=OFF -DGGML_AVX=OFF -DGGML_AVX2=OFF -DGGM
L_AVX512=OFF -DGGML_SSE42=OFF -DGGML_F16C=OFF -DGGML_FMA=OFF
CMAKE_BUILD_TYPE=Release
-- Warning: ccache not found - consider installing it for faster compilation or disable this warning with GGML_CCACHE=OFF
-- CMAKE_SYSTEM_PROCESSOR: x86_64
-- GGML_SYSTEM_ARCH: x86
-- Including CPU backend
-- x86 detected
-- Adding CPU backend variant ggml-cpu:
-- ggml version: 0.9.4
-- ggml commit:  31c511a96
-- Configuring done (0.3s)
-- Generating done (0.1s)
-- Build files have been written to: /home/akshay/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp/build
```

B. Benchmark output (single-thread run).



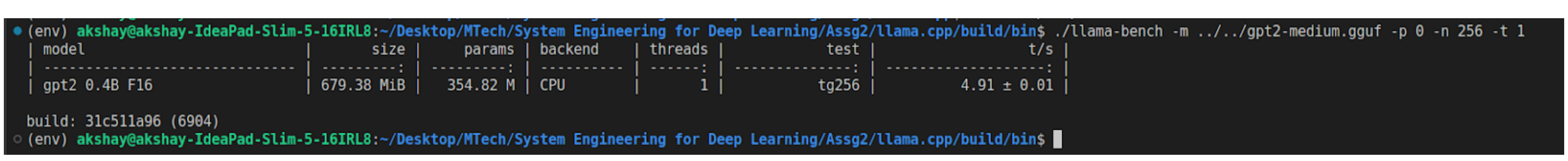
```
nv) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp/build/bin$ ./llama-bench -m ../../gpt2-med
m.gguf -p 0 -n 256 -t 1
model                          |       size |     params | backend    | threads |            test |                  t/s |
------------------------------ | ---------: | ---------: | ---------- | ------: | --------------: | -------------------: |
gpt2 0.4B F16                  | 679.38 MiB |   354.82 M | CPU        |       1 |           tg256 |          4.92 ± 0.02 |

ild: 31c511a96 (6904)
```

Task 4: Default Execution (5 Marks)

A. Steps to build llama.cpp with default settings.

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

B. Benchmark output (single-thread run).

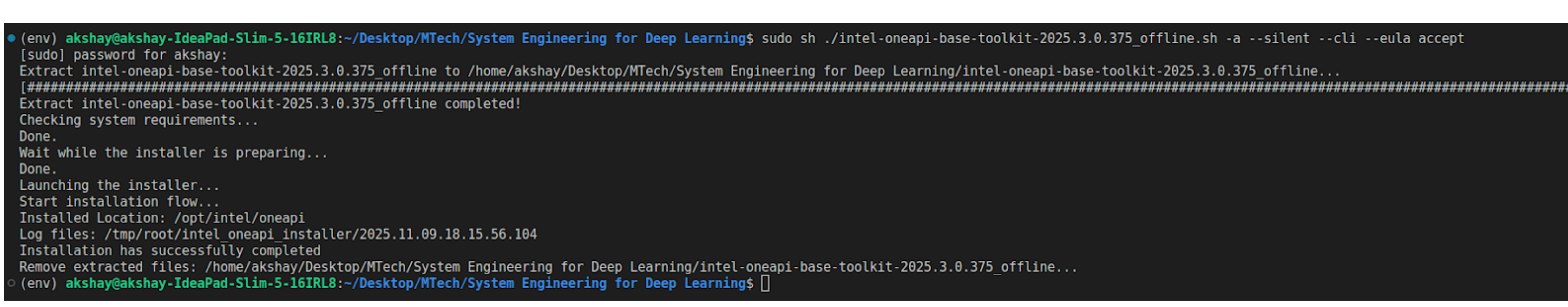


```
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp/build/bin$ ./llama-bench -m ../../gpt2-medium.gguf -p 0 -n 256 -t 1
| model                          |       size |     params | backend    | threads |            test |                  t/s |
| ------------------------------ | ---------: | ---------: | ---------- | ------: | --------------: | -------------------: |
| gpt2 0.4B F16                  | 679.38 MiB |   354.82 M | CPU        |       1 |           tg256 |          4.91 ± 0.01 |

build: 31c511a96 (6904)
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp/build/bin$
```

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

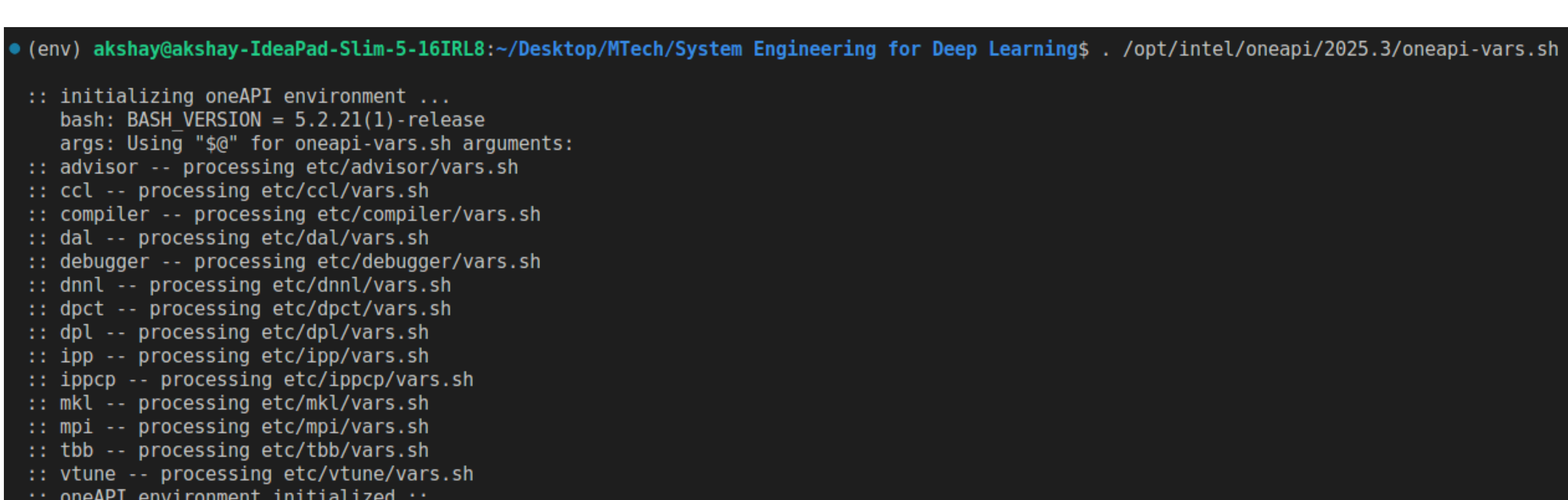
A. Steps to rebuild llama.cpp with Intel MKL support.

- `Install Intel oneAPI by following the instructions given in this document -` [Optimizing and Running LLaMA2 on Intel® CPU]



```
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning$ sudo sh ./intel-oneapi-base-toolkit-2025.3.0.375_offline.sh -a --silent --cli --eula accept
[sudo] password for akshay:
Extract intel-oneapi-base-toolkit-2025.3.0.375_offline to /home/akshay/Desktop/MTech/System Engineering for Deep Learning/intel-oneapi-base-toolkit-2025.3.0.375_offline...
[##########################################################################################################################]
Extract intel-oneapi-base-toolkit-2025.3.0.375_offline completed!
Checking system requirements...
Done.
Wait while the installer is preparing...
Done.
Launching the installer...
Start installation flow...
Installed Location: /opt/intel/oneapi
Log files: /tmp/root/intel_oneapi_installer/2025.11.09.18.15.56.104
Installation has successfully completed
Remove extracted files: /home/akshay/Desktop/MTech/System Engineering for Deep Learning/intel-oneapi-base-toolkit-2025.3.0.375_offline...
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning$
```

- `source /opt/intel/oneapi/2025.3/oneapi-vars.sh`



```
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning$ . /opt/intel/oneapi/2025.3/oneapi-vars.sh

:: initializing oneAPI environment ...
   bash: BASH_VERSION = 5.2.21(1)-release
   args: Using "$@" for oneapi-vars.sh arguments:
:: advisor -- processing etc/advisor/vars.sh
:: ccl -- processing etc/ccl/vars.sh
:: compiler -- processing etc/compiler/vars.sh
:: dal -- processing etc/dal/vars.sh
:: debugger -- processing etc/debugger/vars.sh
:: dnnl -- processing etc/dnnl/vars.sh
:: dpct -- processing etc/dpct/vars.sh
:: dpl -- processing etc/dpl/vars.sh
:: ipp -- processing etc/ipp/vars.sh
:: ippcp -- processing etc/ippcp/vars.sh
:: mkl -- processing etc/mkl/vars.sh
:: mpi -- processing etc/mpi/vars.sh
:: tbb -- processing etc/tbb/vars.sh
:: vtune -- processing etc/vtune/vars.sh
:: oneAPI environment initialized ::
```

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

```
-- Warning: ccache not found - consider installing it for faster compilation or disable this warning with GGML_CCACHE=OFF
-- CMAKE_SYSTEM_PROCESSOR: x86_64
-- GGML_SYSTEM_ARCH: x86
-- Including CPU backend
-- Found OpenMP_C: -fiopenmp (found version "5.1")
-- Found OpenMP_CXX: -fiopenmp (found version "5.1")
-- Found OpenMP: TRUE (found version "5.1")
-- x86 detected
-- Adding CPU backend variant ggml-cpu: -march=native
-- Looking for sgemm_
-- Looking for sgemm_ - found
-- Found BLAS: /opt/intel/oneapi/2025.3/lib/libmkl_intel_lp64.so;/opt/intel/oneapi/2025.3/lib/libmkl_intel_thread.so;/opt/intel/oneapi/2025.3/lib/libmkl_core.so;/opt/intel/oneapi/2025.3/lib/libiomp5.so;-
lm;-ldl
-- BLAS found, Libraries: /opt/intel/oneapi/2025.3/lib/libmkl_intel_lp64.so;/opt/intel/oneapi/2025.3/lib/libmkl_intel_thread.so;/opt/intel/oneapi/2025.3/lib/libmkl_core.so;/opt/intel/oneapi/2025.3/lib/li
biomp5.so;-lm;-ldl
-- Found PkgConfig: /usr/bin/pkg-config (found version "1.8.1")
-- Checking for module 'mkl-sdl'
--   Found mkl-sdl, version 2025.3
-- BLAS found, Includes: /opt/intel/oneapi/mkl/2025.3/lib/pkgconfig/../../include
-- Including BLAS backend
-- ggml version: 0.9.4
-- ggml commit:  31c511a96
-- Found CURL: /usr/lib/x86_64-linux-gnu/libcurl.so (found version "8.5.0")
-- Configuring done (2.4s)
-- Generating done (0.1s)
-- Build files have been written to: /home/akshay/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp/build
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp$
```

- `cmake –build build –config Release`

B. Benchmark output (single-thread run).

```
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp$ ./build/bin/llama-bench -m gpt2-medium.gguf -p 0 -n 256 -t 1
| model                          |       size |     params | backend    | threads |            test |                  t/s |
| ------------------------------ | ---------: | ---------: | ---------- | ------: | --------------: | -------------------: |
| gpt2 0.4B F16                  | 679.38 MiB |   354.82 M | BLAS       |       1 |           tg256 |         26.21 ± 0.53 |

build: 31c511a96 (6904)
```

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

List all relevant floating-point and memory traffic performance counters available on your system.

Floating-point operation counters	**DRAM traffic counters**
1. fp_arith_dispatched.port_0 2. fp_arith_dispatched.port_1 3. fp_arith_dispatched.port_5 4. fp_arith_dispatched.v0 5. fp_arith_dispatched.v1 6. fp_arith_dispatched.v2 7. fp_arith_inst_retired.128b_packed_double 8. fp_arith_inst_retired.128b_packed_single 9. fp_arith_inst_retired.256b_packed_double 10. fp_arith_inst_retired.256b_packed_single 11. fp_arith_inst_retired.4_flops 12. fp_arith_inst_retired.scalar 13. fp_arith_inst_retired.scalar_double 14. fp_arith_inst_retired.scalar_single 15. fp_arith_inst_retired.vector	1. uncore_imc_free_running/data_read 2. uncore_imc_free_running/data_total 3. uncore_imc_free_running/data_write

Tabulate counters with a short description of what each measures.

Category	**Counter Name**	**Description**
Floating-Point Arithmetic (FLOPs)	fp_arith_inst_retired.scalar_single	Counts retired scalar single-precision (32-bit) floating-point arithmetic instructions (SSE/AVX scalar operations).
Floating-Point Arithmetic (FLOPs)	fp_arith_inst_retired.scalar_double	Counts retired scalar double-precision (64-bit) floating-point arithmetic instructions.
Floating-Point Arithmetic (FLOPs)	fp_arith_inst_retired.128b_packed_single	Counts retired 128-bit vector floating-point instructions operating on single-precision data (typically 4 elements).
Floating-Point Arithmetic (FLOPs)	fp_arith_inst_retired.128b_packed_double	Counts retired 128-bit vector floating-point instructions operating on double-precision data (typically 2 elements).
Floating-Point Arithmetic (FLOPs)	fp_arith_inst_retired.256b_packed_single	Counts retired 256-bit vector floating-point instructions operating on single-precision data (typically 8 elements).
Floating-Point Arithmetic (FLOPs)	fp_arith_inst_retired.256b_packed_double	Counts retired 256-bit vector floating-point instructions operating on double-precision data (typically 4 elements).
(Optional / Specialized)	fp_arith_inst_retired.fma	(If available) Counts retired Fused Multiply-Add (FMA) instructions; each represents 2 FLOPs.
(Not used for OI)	fp_arith_inst_retired.vector, fp_arith_inst_retired.scalar, fp_arith_dispatched.*	General or dispatch-side counters; excluded because they don't distinguish vector widths or retired FLOPs.
Memory Traffic (Bytes)	uncore_imc_free_running/data_read/	Counts 64-byte cache lines read from DRAM by the integrated memory controller (IMC). Multiply by 64 for bytes read.
Memory Traffic (Bytes)	uncore_imc_free_running/data_write/	Counts 64-byte cache lines written to DRAM by the IMC. Multiply by 64 for bytes written.
(Alternative / Derived)	uncore_imc_free_running/data_total/	Counts total 64-byte data transactions (reads + writes) through the IMC; used as a direct estimate of total memory bytes.

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

A. Submit performance counter values for each build variant (Tasks 3, 4, 5).

Please see Table 7.1 in the next subsection for performance counter values with OI derivations.

B. Derive and report Operation Intensity (OI) for each case.

I investigated whether the compiled LLaMA benchmark binary makes use of FMA (fused multiply-add) instructions, because an FMA performs two floating-point operations (a multiply and an add) per element and therefore affects the FLOP count. I disassembled the llama-bench binary using **`objdump`** and searched for FMA mnemonics. The disassembly clearly shows **`vfmadd*`** and **`vfmsub*`** instructions (for example vfmadd213ss and vfnmadd231ss inside SVML routines such as __svml_scos_data_internal), confirming that FMAs are indeed present in the binary. Please see the disassembly results in the image below.

```
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp$ objdump -d ./build/bin/llama-bench | grep -Ei 'vfmadd|vfmsub|vfnmadd|vfnmsub' | head
  4575c7:       c4 e2 41 99 2d b0 27    vfmadd132ss 0x327b0(%rip),%xmm7,%xmm5        # 489d80 <__svml_scos_data_internal+0x1580>
  4575e9:       c4 e2 49 ad fc          vfnmadd213ss %xmm4,%xmm6,%xmm7
  4575ee:       c4 e2 49 bd 3d 49 25    vfnmadd231ss 0x32549(%rip),%xmm6,%xmm7        # 489b40 <__svml_scos_data_internal+0x1340>
  4575fc:       c4 e2 49 bd 3d 7b 25    vfnmadd231ss 0x3257b(%rip),%xmm6,%xmm7        # 489b80 <__svml_scos_data_internal+0x1380>
  457615:       c4 e2 49 a9 3d e2 26    vfmadd213ss 0x326e2(%rip),%xmm6,%xmm7        # 489d00 <__svml_scos_data_internal+0x1500>
  45761e:       c4 e2 49 a9 3d 99 26    vfmadd213ss 0x32699(%rip),%xmm6,%xmm7        # 489cc0 <__svml_scos_data_internal+0x14c0>
  457627:       c4 e2 49 a9 3d 90 25    vfmadd213ss 0x32590(%rip),%xmm6,%xmm7        # 489bc0 <__svml_scos_data_internal+0x13c0>
  457634:       c4 e2 41 a9 f5          vfmadd213ss %xmm5,%xmm7,%xmm6
  457875:       c4 62 39 ab cf          vfmsub213ss %xmm7,%xmm8,%xmm9
  45787a:       c4 62 39 b9 0d bd 97    vfmadd231ss 0x297bd(%rip),%xmm8,%xmm9        # 481040 <_IO_stdin_used+0x40>
```

For the LLaMA benchmark configuration in Task 3(parallelism disabled build), I measured the floating-point operation intensity using hardware performance counters. Since my processor does not expose a dedicated `fp_arith_inst_retired.fma` event, I cannot directly count fused multiply-add (FMA) instructions separately from other arithmetic operations. Instead, I evaluated two models for this build.

In the **lower-bound model**, I assumed that each floating-point arithmetic instruction corresponds to one FLOP per element (i.e., I ignored the additional FLOP contributed by FMAs). Under this assumption, I obtained a total of 9.42×10^{11} FLOPs and 1.36×10^{12} bytes of DRAM traffic, which yields an operational intensity of approximately **0.691 FLOPs/byte**.

In the **upper-bound model**, I assumed that the vector floating-point instructions are predominantly FMAs and therefore account for 2 FLOPs per element. With this assumption, the total FLOPs increase slightly to 9.44×10^{11}, while the measured DRAM traffic is 1.30×10^{12} bytes, giving an operational intensity of approximately **0.725 FLOPs/byte**. The difference between the two models is about **5%**, which I interpret as a reasonable bound on the uncertainty due to unknown FMA usage.

Importantly, both values (0.691 and 0.725 FLOPs/byte) remain below the machine's crossover intensity $I^* \approx 0.77$ FLOPs/byte(derivation shown here). Therefore, regardless of the exact FMA fraction, this LLaMA configuration is classified as **memory-bound** in the Roofline model. In the subsequent plots, I use the FMA-heavy value (upper-bound model value) as a slightly more optimistic estimate for all the builds.

So essentially the formula used to derive operational intensity with FMA assumption is,

$$\textbf{flops}_{\textbf{total}} = \text{fp\_arith\_inst\_retired.scalar\_single} + \text{fp\_arith\_inst\_retired.scalar\_double}$$

$$+ (\text{fp\_arith\_inst\_retired.128b\_packed\_single} + \text{fp\_arith\_inst\_retired.128b\_packed\_double}) * 2.0 * \text{fma\_factor}$$

$$+ (\text{fp\_arith\_inst\_retired.256b\_packed\_single} + \text{fp\_arith\_inst\_retired.256b\_packed\_double}) * 4.0 * \text{fma\_factor}$$

where fma_factor = 2

$$\textbf{bytes}_{\textbf{total}} = \text{uncore\_imc\_free\_running/data\_read/} + \text{uncore\_imc\_free\_running/data\_write/}$$

$$\textbf{operational intensity} = \textbf{flops}_{\textbf{total}} / \textbf{bytes}_{\textbf{total}}$$

Note: Running perf stat command along with deriving all the values populated in the following tables have been done using a small python helper script - oi_perf.py. Sample output of running the script is given below for reference.

```
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2/llama.cpp$ sudo python3 oi_perf.py --assume-fma -- ./build/bin/llama-bench -m ./gpt2-medium.gguf -p 0 -n 256 -
t 1
Running perf command:
 perf stat -x , --no-big-num -a -e fp_arith_inst_retired.scalar_single,fp_arith_inst_retired.scalar_double,fp_arith_inst_retired.128b_packed_single,fp_arith_inst_retired.128b_packed_double,fp_arith_inst_retired.
256b_packed_single,fp_arith_inst_retired.256b_packed_double,uncore_imc_free_running/data_read/,uncore_imc_free_running/data_write/,uncore_imc_free_running/data_total/,mem_inst_retired.all_loads,mem_inst_retired.
all_stores -- env LD_LIBRARY_PATH=/opt/intel/oneapi/2025.3/lib:/opt/intel/oneapi/installer/lib ./build/bin/llama-bench -m ./gpt2-medium.gguf -p 0 -n 256 -t 1

=============== Operation Intensity Report ===============
Command: ./build/bin/llama-bench -m ./gpt2-medium.gguf -p 0 -n 256 -t 1
Injected LD_LIBRARY_PATH:
  /opt/intel/oneapi/2025.3/lib:/opt/intel/oneapi/installer/lib

Assume FMA for packed ops: YES (×2 per element)

FLOP counters (retired):
  fp_arith_inst_retired.scalar_single           471,367,565,038
  fp_arith_inst_retired.scalar_double           469,526,290,240
  fp_arith_inst_retired.128b_packed_single      1,750,881,918
  fp_arith_inst_retired.128b_packed_double      311,666,462
  fp_arith_inst_retired.256b_packed_single      179,136,330
  fp_arith_inst_retired.256b_packed_double      0

Memory traffic source: IMC(read+write)
  uncore_imc_free_running/data_read/            1,551,179,542,691.840088 bytes
  uncore_imc_free_running/data_write/           104,159,395,184.639999 bytes
  Total Bytes                                   1,655,338,937,876.479980 bytes

Totals:
  FLOPs      = 950,577,139,438.000000
  Bytes      = 1,655,338,937,876.479980
  OI         = FLOPs / Bytes = 0.574249 FLOPs/Byte
  Time       = 525.891391 seconds (wall clock, including perf)
  GFLOPs/s   = 1.807554
  GB/s       = 3.147682
==========================================================
```

Table 7.1 - Performance counter values with OI			
Performance counter	No parallelism build(Task 3)	Default build(Task 4)	Intel MKL enabled build(Task 5)
fp_arith_inst_retired.scalar_single	471,367,565,038	623,287,623	1,109,501,566
fp_arith_inst_retired.scalar_double	469,526,290,240	90,455,902	128,843,891
fp_arith_inst_retired.128b_packed_single	1,750,881,918	1,562,941,491	1,569,807,878
fp_arith_inst_retired.128b_packed_double	311,666,462	13,224	2,666,180
fp_arith_inst_retired.256b_packed_single	179,136,330	119,223,203,428	119,171,682,386
fp_arith_inst_retired.256b_packed_double	0	0	116,754
uncore_imc_free_running/data_read/	1,551,179,542,691.840088 bytes	978,746,028,851.199951 bytes	992,655,148,318.719971 bytes
uncore_imc_free_running/data_write/	104,159,395,184.639999 bytes	3,330,340,290.560000 bytes	7,631,640,985.600000 bytes
Total Bytes	1,655,338,937,876.479980	982,076,369,141.760010	1,000,286,789,304.319946
Total FLOPs	950,576,729,700.000000	960,751,189,809.000000	960,902,634,809.000000
OI	0.574249 FLOPs/Byte	0.978286 FLOPs/Byte	0.960627 FLOPs/Byte
Time taken	525.891391 seconds	71.594758 seconds	76.412846 seconds
Throughput (GFLOPs/s)	1.807554	13.419295	12.575145
Bandwidth (GB/s)	3.147682	13.717155	13.090558

C. Include peak memory bandwidth and compute capacity of your system.

1. Architectural Assumptions for Compute Peak

Before performing any measurements, I made the following standard microarchitectural assumptions about the floating-point capabilities of Intel Raptor Lake mobile CPUs (13th Gen), based on publicly known microarchitecture details:

1. All cores support:
   - **AVX2** vector instructions
   - **FMA3** fused multiply-add operations
2. I confirmed this by checking for the **`avx2`** and **`fma`** flags in **`/proc/cpuinfo.`**
3. FLOPs per vector FMA instruction:
   A 256-bit AVX2 FMA on FP32 operates on **8 lanes** and performs a multiply and an add:
   8 elements × 2 flops/element = 16 FLOPs per FMA
4. FLOPs per cycle assumptions:
   - P-cores sustain **2 FMA units per cycle**, giving
     32 FLOPs/cycle (FP32)
   - E-cores typically sustain **1 FMA unit per cycle**, giving
     16 FLOPs/cycle (FP32)

Given the hybrid architecture, the theoretical compute peak (FP32) would be:

$$P_{theoretical} = N_p * f_p * 32 + N_e * f_e * 16$$

where:

- N_p = 6 P-cores
- N_e = 8 E-cores
- f_p and f_e are frequencies in GHz

However, as effective sustained frequencies under heavy load are lower than the maximum turbo frequencies, instead of relying on assumed frequencies, I decided to **directly measure sustained peak compute performance**, which provides a more realistic value for Roofline modeling.

2. Measured Compute Peak P_{max}

To determine the sustained practical compute peak, I executed a large FP32 SGEMM(Single-precision General Matrix Multiply) operation using NumPy's matrix multiplication, which internally calls the system BLAS library (OpenBLAS/MKL depending on the build). I wrote a script(sgemm.sh) that does a 4096×4096 matrix multiply, which is large enough to saturate the floating-point pipelines and minimize overhead.

I ran the benchmark on **all 20 logical CPUs** with:

```
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2$ ./sgemm.sh
All-threads SGEMM GFLOPs/s: 78.74536935505454  time: 1.745359182357788  threads: 20
```

From this result, I take the **sustained compute peak** as:

$$\mathbf{P_{max} = 78.7\ GFLOPs/s}$$

This value reflects real thermal throttling, BLAS efficiency and architectural limitations under sustained load, making it appropriate for Roofline modeling.

3. Assumptions for Memory Bandwidth Estimation

To estimate the peak sustained memory bandwidth of the system:

1. I used a **large streaming memory copy**(stream_test.sh), which behaves similarly to the STREAM benchmark and is known to reach near-peak DRAM bandwidth on modern architectures.
2. I instrumented the workload using **Intel uncore IMC free-running counters**:
   - `uncore_imc_free_running/data_read/`
   - `uncore_imc_free_running/data_write/`
3. These counters report **total traffic through the DRAM memory controller**, which is the correct source for determining memory bandwidth for Roofline modeling.

I ensured the working set was significantly larger than cache (≈600 MB) so that the operation was truly DRAM-bound.

4. Measured Memory Bandwidth B_{max}

The stream copy test produced the following measurements:

```
(env) akshay@akshay-IdeaPad-Slim-5-16IRL8:~/Desktop/MTech/System Engineering for Deep Learning/Assg2$ ./stream_test.sh
copy seconds: 0.5233404636383057
26790.45,MiB,uncore_imc_free_running/data_read/,48511267938,100.00,,
24305.76,MiB,uncore_imc_free_running/data_write/,48511258553,100.00,,
```

We can convert MiB to bytes using:

$$1\ \text{MiB} = 1{,}048{,}576\ \text{bytes}$$

So, total bytes transferred:

$$\text{Bytes}_{total} \approx 53.58\times10^{9}\ \text{bytes}$$

Memory bandwidth (bytes per second):

$$B_{max} = 53.58\times10^{9} / 0.52334 \approx 1.02\times10^{11}\ \text{bytes/s}$$

Convert to GB/s using 1 GB=10^9 bytes

$$B_{max} \approx 102.4\ \text{GB/s}$$

Thus the sustained memory bandwidth is:

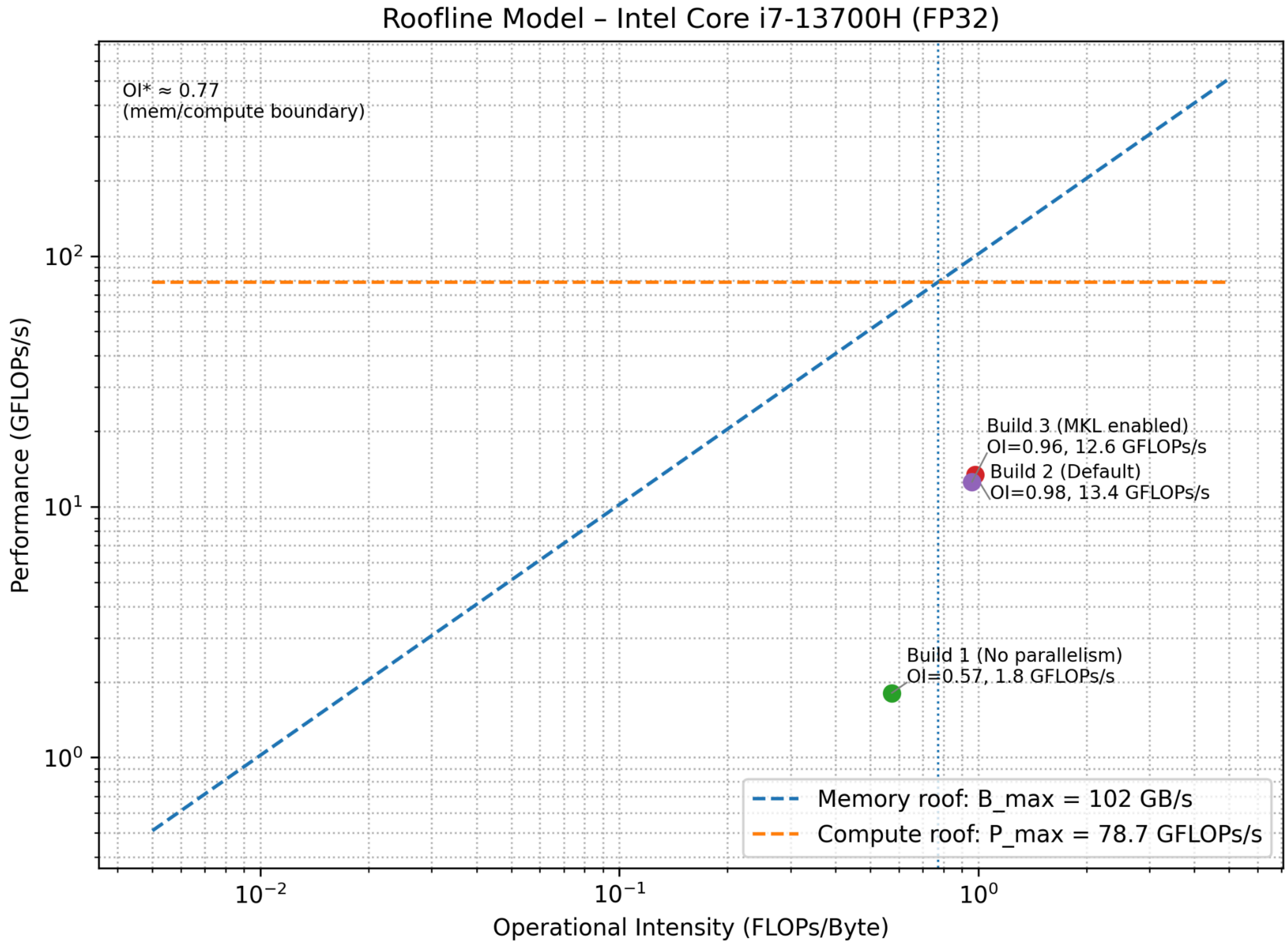
$$\mathbf{B_{max} = 102\ GB/s}$$

5. Crossover Operational Intensity

The operational intensity where the Roofline transitions from memory-bound to compute-bound is:

$$\mathbf{I^{*} = P_{max}/B_{max} = 78.7 / 102.4 \approx 0.77\ FLOPs/byte}$$

D. Insert Roofline analysis plot overlaying all variants.



E. Short commentary on memory-bound vs compute-bound performance.

The first build (no parallelism) achieved an operational intensity of 0.574 FLOPs/Byte, which is significantly below the crossover point. Consequently, I classify this configuration as **memory-bound**, because its performance is restricted by the rate at which data can be supplied from memory. The relatively low achieved performance of 1.81 GFLOPs/s further indicates that the processor spends most of its execution time stalling on memory accesses rather than performing floating-point arithmetic.

The second build (default optimized) attained an operational intensity of 0.978 FLOPs/Byte, which exceeds the crossover threshold. This places the computation in the **compute-bound** region. Here, the arithmetic workload per byte of memory traffic is sufficiently high for the CPU's floating-point units to become the primary bottleneck. The significant improvement to 13.42 GFLOPs/s reflects this shift: rather than waiting on memory, the kernel now stresses the processor's compute resources.

The third build (MKL-enabled) exhibited a similar operational intensity of 0.960 FLOPs/Byte and a performance level of 12.58 GFLOPs/s. Since its OI remains above crossover point, this build is also **compute-bound**. Although MKL typically provides high performance, the particular structure of the evaluated workload does not allow it to deliver higher arithmetic throughput than the default optimized version. This suggests that the kernel's computational characteristics, rather than the specific library implementation, dominate performance at this point on the roofline.

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

A. Benchmark logs and performance counter outputs for all tested thread counts.

Table 8.1.1 - Performance counter values with thread scaling			
Performance counter	**2 Threads**	**4 Threads**	**8 Threads**
fp_arith_inst_retired.scalar_single	1,113,984,878	1,113,127,470	689,331,910
fp_arith_inst_retired.scalar_double	125,248,330	126,918,122	77,237,152
fp_arith_inst_retired.128b_packed_single	1,572,952,286	1,570,183,307	1,242,824,410
fp_arith_inst_retired.128b_packed_double	5,008,194	4,303,659	2,165,503
fp_arith_inst_retired.256b_packed_single	119,163,613,668	118,999,205,661	95,697,220,784
fp_arith_inst_retired.256b_packed_double	0	180,224	133,562
uncore_imc_free_running/data_read/	983,448,965,611.520020 bytes	995,898,572,144.640015 bytes	1,021,982,749,491.199951 bytes
uncore_imc_free_running/data_write/	5,798,499,450.880000 bytes	6,189,807,042.560000 bytes	18,243,943,137.279999 bytes

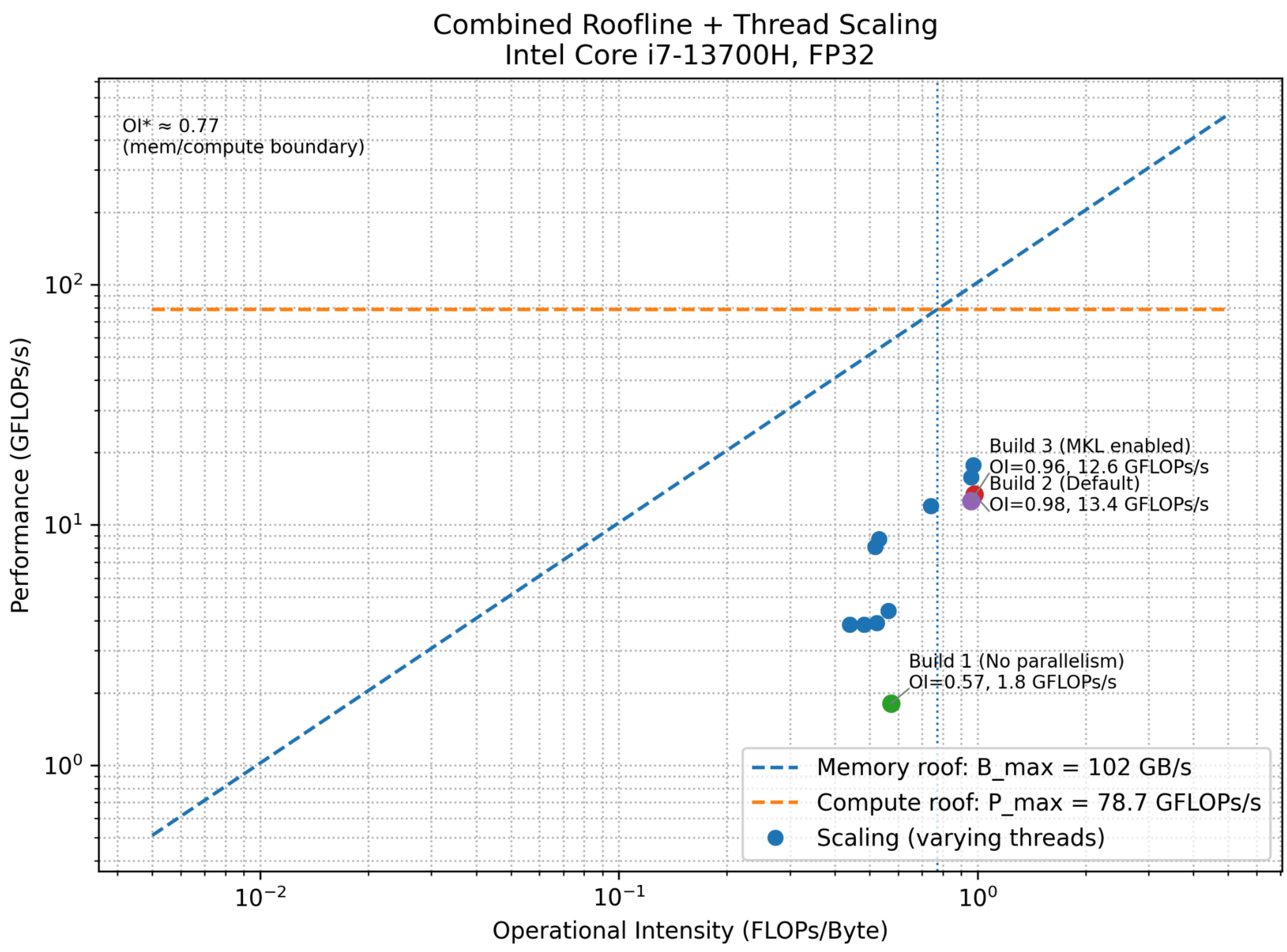
Table 8.1.1 - Performance counter values with thread scaling			
Total Bytes	989,247,465,062.400024 bytes	1,002,088,379,187.200073 bytes	1,040,226,692,628.479980 bytes
Total FLOPs	960,859,984,472.000000	959,533,080,536.000000	771,325,363,482.000000
OI	0.971304 FLOPs/Byte	0.957533 FLOPs/Byte	0.741497 FLOPs/Byte
Time taken	54.168406 seconds	60.857405 seconds	64.487407 seconds
Throughput (GFLOPs/s)	17.738384	15.766908	11.960868
Bandwidth (GB/s)	18.262444	16.466170	16.130695

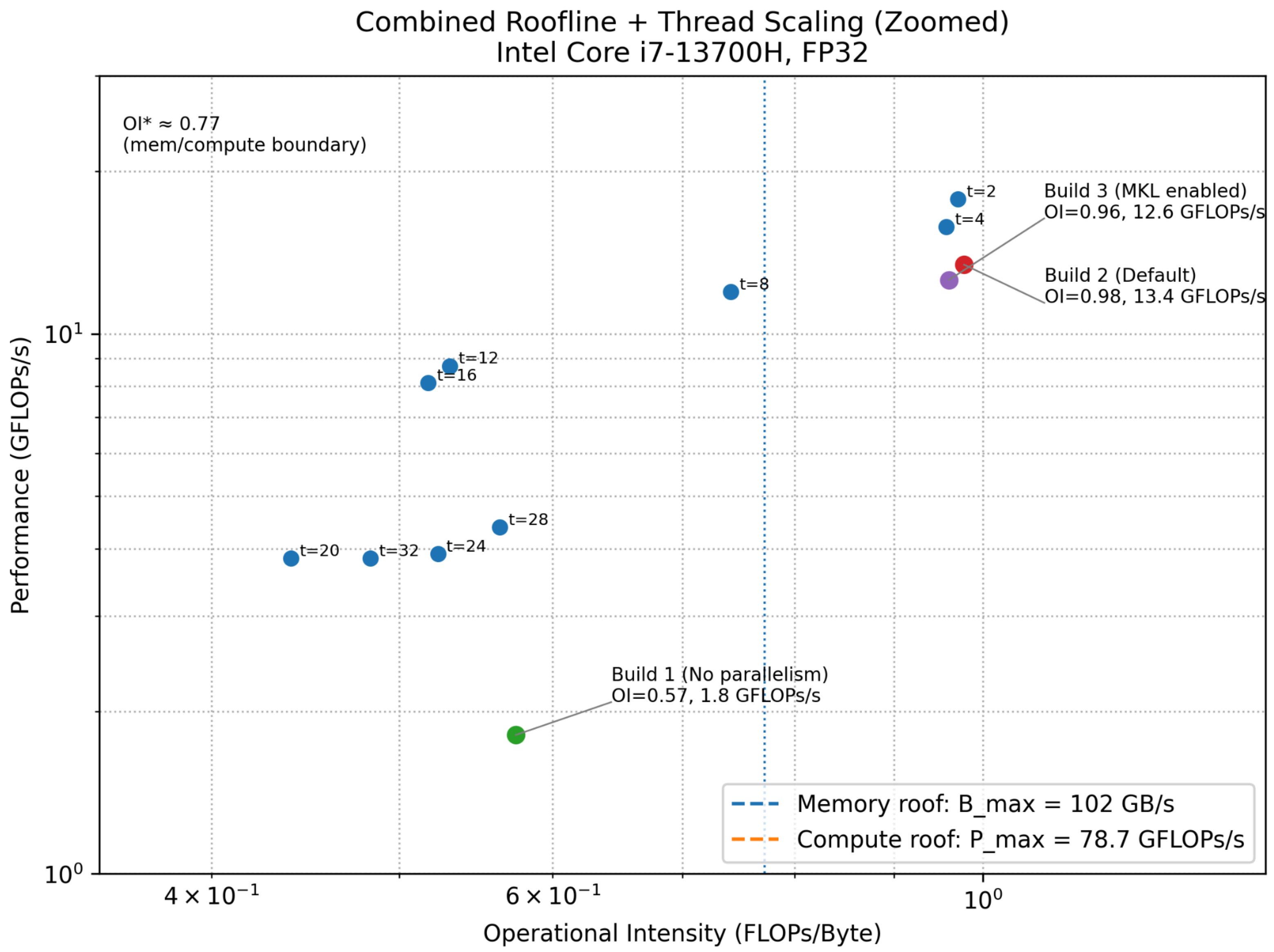
Table 8.1.2 - Performance counter values with thread scaling			
Performance counter	**12 Threads**	**16 Threads**	**20 Threads**
fp_arith_inst_retired.scalar_single	1,004,053,392	1,007,529,647	813,986,547
fp_arith_inst_retired.scalar_double	118,615,254	160,279,355	169,498,937
fp_arith_inst_retired.128b_packed_single	934,479,016	909,653,237	974,618,370
fp_arith_inst_retired.128b_packed_double	143,518	18,714,662	1,096,854
fp_arith_inst_retired.256b_packed_single	70,222,589,127	65,769,013,205	71,348,786,975
fp_arith_inst_retired.256b_packed_double	0	180,224	11,836
uncore_imc_free_running/data_read/	1,038,137,142,804.479980 bytes	1,018,586,684,456.959961 bytes	1,232,332,138,741.760010 bytes
uncore_imc_free_running/data_write/	29,174,760,734.720001 bytes	7,729,955,471.360000 bytes	77,424,146,513.919998 bytes
Total Bytes	1,067,311,903,539.199951 bytes	1,026,316,639,928.319946 bytes	1,309,756,285,255.679932 bytes
Total FLOPs	566,641,871,798.000000	531,034,828,030.000000	575,676,736,868.000000
OI	0.530906 FLOPs/Byte	0.517418 FLOPs/Byte	0.439530 FLOPs/Byte
Time taken	65.010085 seconds	65.473432 seconds	149.931798 seconds
Throughput (GFLOPs/s)	8.716215	8.110692	3.839591
Bandwidth (GB/s)	16.417636	15.675315	8.735681

Table 8.1.3 - Performance counter values with thread scaling			
Performance counter	**24 Threads**	**28 Threads**	**32 Threads**
fp_arith_inst_retired.scalar_single	1,100,489,466	682,949,362	790,824,888
fp_arith_inst_retired.scalar_double	144,821,369	71,891,999	104,937,144
fp_arith_inst_retired.128b_packed_single	1,036,130,511	968,035,010	981,174,226
fp_arith_inst_retired.128b_packed_double	9,294,220	112,994	2,327,960
fp_arith_inst_retired.256b_packed_single	74,770,483,698	74,464,620,623	72,180,891,919
fp_arith_inst_retired.256b_packed_double	180,224	0	180,224
uncore_imc_free_running/data_read/	1,135,310,033,387.520020 bytes	1,060,175,515,484.160034 bytes	1,169,965,354,844.159912 bytes
uncore_imc_free_running/data_write/	17,835,973,672.959999 bytes	5,544,293,171.200000 bytes	35,234,974,269.440002 bytes
Total Bytes	1,153,146,007,060.479980 bytes	1,065,719,808,655.359985 bytes	1,205,200,329,113.599854 bytes
Total FLOPs	603,592,321,135.000000	600,344,398,361.000000	582,278,347,920.000000
OI	0.523431 FLOPs/Byte	0.563323 FLOPs/Byte	0.483138 FLOPs/Byte
Time taken	154.158961 seconds	137.029044 seconds	149.931798 seconds
Throughput (GFLOPs/s)	3.915389	4.381147	3.839591

Table 8.1.3 - Performance counter values with thread scaling			
Bandwidth (GB/s)	7.480240	7.777328	8.735681

B. Roofline plot showing scaling behavior.





C. Discussion on how close MKL build approaches peak compute throughput.

In this task, I evaluated how the performance characteristics of the LLaMA benchmark change as I scale the number of threads. I conducted multiple runs with thread counts of 1, 2, 4, 8, 12, 16, 20, 24, 28, and 32, and for each configuration I collected hardware performance counters using **perf**. These counters allowed me to compute the achieved FLOPs, DRAM traffic, and Operational Intensity (OI) for each run. I then extended the roofline plot to visualize the scaling behavior across the tested thread counts.

To interpret the results, I compared each measured OI value against the machine's crossover intensity OI ≈ 0.77 FLOPs/Byte, which I previously derived from the sustained compute peak (≈78.7 GFLOPs/s) and sustained memory bandwidth (≈102 GB/s). This threshold effectively separates compute-bound behavior (OI > OI*) from memory-bound behavior (OI < OI*).

At low thread counts (specifically 2 and 4 threads), I observed OI values in the range of 0.96–0.97 FLOPs/Byte, which exceed the crossover intensity. These configurations therefore operate in the **compute-bound region**, where arithmetic throughput is the primary limiting factor. Correspondingly, these runs achieved the highest performance, in the range of 15–18 GFLOPs/s. At this scale, the working set fits comfortably within the private L2 and shared L3 caches, and data reuse is high, allowing the floating-point units to remain well utilized.

However, as I increased the thread count to 8, OI dropped to approximately 0.74 FLOPs/Byte, placing the kernel near the roofline knee where execution begins transitioning from compute-limited to memory-limited behavior. The measured performance also decreased noticeably, indicating the onset of increased cache-level interference and more frequent LLC and DRAM accesses.

For thread counts of 12 and above, the OI values fell sharply into the range of 0.44–0.56 FLOPs/Byte, which is well below the crossover point. These configurations clearly operate in the **memory-bound region**. In this regime, performance is constrained by DRAM bandwidth rather than by computational throughput. As more cores become active, contention in the shared L3 cache increases, data reuse degrades, and the volume of DRAM traffic grows faster than the number of useful floating-point operations. As a result, increasing thread count no longer improves performance, and in some cases even reduces it due to bandwidth saturation and memory-system queuing effects.

At the highest thread counts (20–32 threads), I observed both low OI and consistently poor performance (≈3.8–4.4 GFLOPs/s). This behavior aligns with the expected limitations of the underlying Intel hybrid architecture, where E-core clusters share smaller L2 slices and contend more aggressively for shared resources. At such high concurrency levels, the DRAM controller is fully saturated, and further increases in thread count yield no additional throughput.

Finally, I incorporated all measured OI and performance points into the roofline plot. The resulting visualization clearly illustrates a downward-sloping trend as thread count increases. The points move leftward (due to increasing memory traffic) and downward (due to decreasing arithmetic efficiency), ultimately approaching the memory-bandwidth roof. This progression vividly captures the transition from compute-bound execution at small thread counts to memory-bound execution at moderate and large thread counts.

Overall, the thread-scaling analysis demonstrates that this workload exhibits strong memory-boundedness beyond 8 threads. The roofline model provides a coherent explanation for the observed performance trends and offers a clear basis for understanding why additional threads fail to deliver proportional speedups once DRAM bandwidth becomes the dominant bottleneck.

Note: All the scripts used for completion of this assignment have been uploaded to this github repository - CS6886W_Assg2-GPT_2_Performance_Analysis.