COMP3230 Principles of Operating Systems

Programming Assignment Two
Due date: November 17 , 2024, at 23:59

Total 12 points – RC2

Programming Exercise – Accelerate LLM Inference using Multi-Threading

Objectives

1. An assessment task related to ILO 4 [Practicability] – "demonstrate knowledge in applying system software and tools available in the modern operating system for software development".
2. A learning activity related to ILO 2.
3. The goals of this programming exercise are:
   - to have hands-on practice in designing and developing multi-threading programs;
   - to learn how to use POSIX Pthreads (and Semaphore) libraries to create, manage, and coordinate multiple threads in a shared memory environment;
   - to design and implement synchronization schemes for multithreaded processes using semaphores, or mutex locks and condition variables.

Tasks

Optimizing the throughput of GPTs is an important topic. Similar to other neural networks, GPT and its variations utilize matrix-vector-multiplication, or called fully-connected/linear layer in Deep Learning, to apply parameters learned. Meanwhile, GPT leverages multi-head attention, a mechanism to adopt important information from history tokens. Thus, to accelerate GPT and get a faster response, it's critical to have faster matrix-vector-multiplication and faster multi-head attention computation. Given the non-sequential property of the two algorithms, parallel computing based on multi-threading is usually considered helpful.

Following PA1, we use Llama3, an open-source variation of GPT for this assignment. A single-thread C implementation of the inference program, named **`seq.c`**, is provided as the starting point of your work. You need to use POSIX Pthreads with either the semaphore or (mutex_lock + condition variable) to implement a multi-threading version of the inference program, which parallelizes both matrix-vector-multiplication and multi-head attention functions. This multi-threading version shall significantly accelerate the inference task of the Large Language Model. Moreover, to reduce the system workload in creating multiple threads, you need to reuse threads by formulating them into a thread pool.

Acknowledgement: The inference framework used in this assignment is based on the open-source project llama2.c by Andrej Karpathy. The LLM used in this assignment is based on SmolLM by HuggingfaceTB. Thanks open-source!

GPT-based Large Language Model

At high-level, GPT is a machine that can **generate** words **one by one** based on **previous words** (also known as prompts), and Figure 1a illustrates the basic workflow of GPT on generating "How are you":

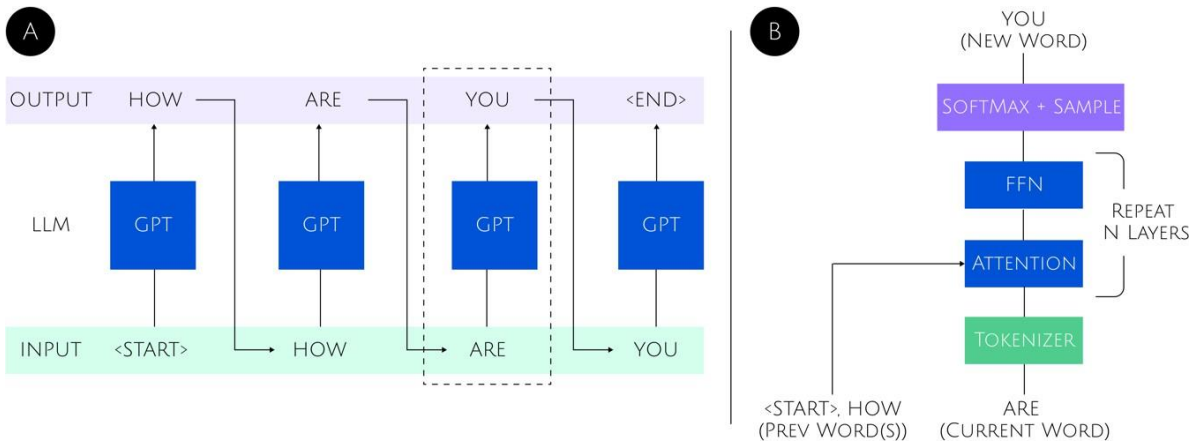

Figure 1. GPT Insight. a) GPT generates text one by one, and each output is the input of the next generation. b) GPT has four major components: Tokenizer turns word (string) into a vector, Softmax + Sample gives the next token, and each layer has Attention and FFN (Feed-Forward Network), consisting of many Matrix-Vector-Multiplication

Figure 1b showcases the inference workflow of each word like “You” in “How are you”: First, words are transformed into token embeddings using the tokenizer, which is essentially a (python) dictionary that assigns a unique vector to each word. Embedding vectors go through multiple layers, which involve three steps.

- The first step is Multi-Head Attention, where the model first calculates attention scores based on the cosine similarity between the current word's query embedding and embeddings of previous words (keys). Then weighted average value embeddings are used to formulate output.
- The second step is a feed-forward network (FFN) that applies more learnable parameters through Matrix-Vector-Multiplication.
- The third step is positional embedding, which takes into account the ordering of words in natural language by adding positional information by RoPE (not required in this assignment).

After going through all the layers, the embeddings are classified to generate a specific word as the output. This involves using a softmax function to convert the embeddings into a probability distribution, and randomly sample a word from the distribution.

Task 1: Matrix-Vector-Multiplication

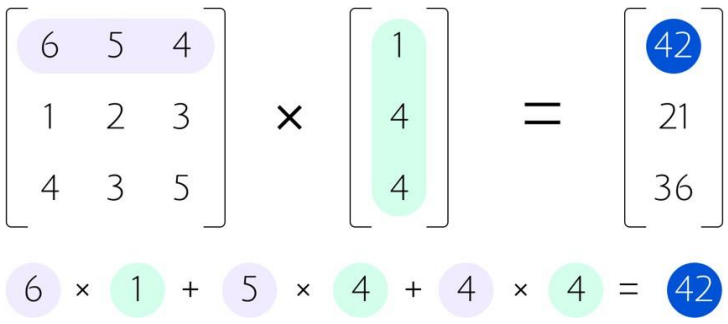

Figure 2. Matrix-Vector-Multiplication Algorithm.

As shown in Figure 2, Matrix-Vector-Multiplication can be illustrated as two iterations:

For Each Row ***i***
    For Column ***j***, accumulate `Matrix[i][j] * Vector[j]` to `Out[i]`

More specifically, a sample single-thread C implementation is shown below:

```c
void mat_vec_mul(float* out, float* vec, float* mat, int col, int row) {
    for (int i = 0; i < row; i++) { // for each row i
        float val = 0.0f;
        for (int j = 0; j < col; j++) // for each column j
            val += mat[i * col + j] * vec[j]; // mat[i * col + j] := mat[i][j]
        out[i] = val;
    }
}
```

Your 1st task in this assignment is to parallelize the outer iteration (at the 2nd line) by allocating blocks of rows to threads. More specifically, in the case of a Matrix with d rows and n threads working on the computation, assuming that d is divisible by n, the k-th thread will handle the rows from $[k \times {}^{d}/_{n}]$ to $[(k+1) \times {}^{d}/_{n} - 1]$. To illustrate, if we have a 6-row matrix with 2 threads, the 0th thread will handle rows 0 to 2, while the 1st thread will handle rows 3 to 5. If d is not divisible by n, we can assign $n-1$ threads with $\lceil {}^{d}/_{n} \rceil$ rows, while the last thread is assigned with the remaining rows. More explanation on such a design can be found in Appendix a. Parallel Checking.

In this assignment, the model used is quantized, so there is a slight difference with the above C code but the workflow is still the same.

Task 2: Multi-Head Attention

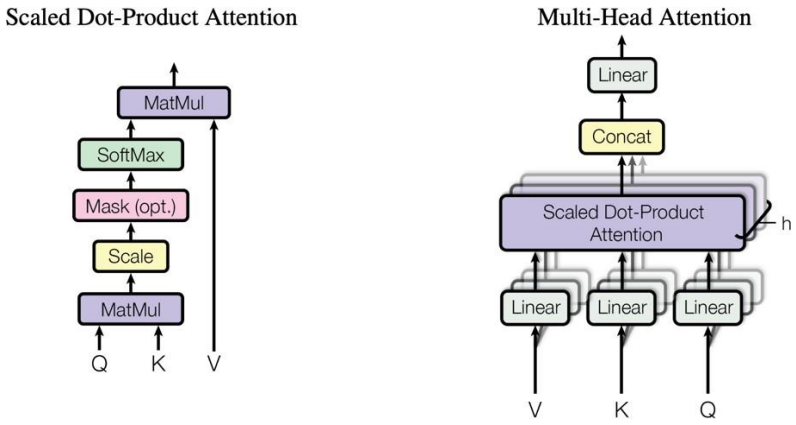

Figure 3. Attention and Multi-Head Attention Algorithm. The computation of each head is separated.

As shown in Figure 3, Multi-Head Attention can be illustrated by the following iterations:

For Each Head ***h***,
  For Each Timestep ***t***,
    For head item ***i***, accumulate `q[h][i] * keys[h][t][i]` to `score[h][t]`
  SoftMax(`score`)
  For Each Timestep ***t***,
    For head item ***i***, `out[h][i] += score[h][t] * values[h][t][i]`

More specifically, a sample single-thread C implementation is shown below:

```
void multi_head_attn(float* out, float* q, float* key_cache, float* value_cache,
float* att, int seq_len, int n_heads, int head_size, int kv_dim, int kv_mul) {
    for (int h = 0; h < n_heads; h++) {       // iterate over all heads, PARALLEL THIS LOOP
        float* head_q = q + h * head_size;    // query vector for this head
        float* head_att = att + h * seq_len; // attention scores for this head
        for (int t = 0; t <= pos; t++) {      // iterate over all timesteps
            // get the key vector for this head and at this timestep
            float* head_k = key_cache + t * kv_dim + (h / kv_mul) * head_size;
            float score = 0.0f;
            for (int i = 0; i < head_size; i++)
                score += head_q[i] * head_k[i]; // attention score := q dot k
            score /= sqrtf(head_size); // normalize by head_size
            head_att[t] = score;       // save to the attention buffer
        }
        softmax(head_att, pos + 1); // THREADS-SAFE SoftMax to normalize scores to weight
        float* head_out = out + h * head_size; // out vector for this head
        memset(head_out, 0, head_size * sizeof(float)); // clear buffer
        for (int t = 0; t <= pos; t++) {
            // get the value vector for this head and at this timestep
            float* head_v = value_cache + t * kv_dim + (h / kv_mul) * head_size;
            float a = head_att[t]; // attention weight for this timestep
            for (int i = 0; i < head_size; i++)
                head_out[i] += a * head_v[i]; // accumulate the weighted sum to head out
        }
    }
}
```

Though looks complicated, it's worth noticing that the computation involved for each head ***k*** is completely independent of other heads.

Your 2nd task in this assignment is to parallelize the head-iteration (in 3rd line of the above sample code) by assigning blocks of heads to different threads. More specifically, consider a model with ***h*** heads and ***n*** threads. If ***h*** is divisible by ***n***, then ***k***-th thread will handle heads from $[k \times {}^{h}/_{n}]$ to $[(k+1) \times {}^{h}/_{n} - 1]$. And if ***h*** is not divisible by ***n***, we can assign $n-1$ threads with $\lceil {}^{h}/_{n} \rceil$ heads, and the last thread handles the remaining heads. For example, our model has 9 heads, and with 4 threads, they will handle 0-2 (1st thread), 3-5 (2nd thread), 6-8 (3rd thread) and the last thread shall handle no heads.

Note: Due to MQA, the no. of heads for a query might not be equal to the no. of heads for key / value, but this is already handled correctly within the inner loop, just stick to `n_heads`.

Task 3: Thread Pool

Moreover, to reduce the performance overhead of frequent thread creation and cancellation, your 3rd task is to create one set of ***N*** threads and reuse them for all `mat_vec_mul()` and `multi_head_attn()` function calls, instead of creating ***N*** threads for each `mat_vec_mul()` or `multi_head_attn()` call. One popular method is based on synchronization using a thread pool as shown in Figure 4.

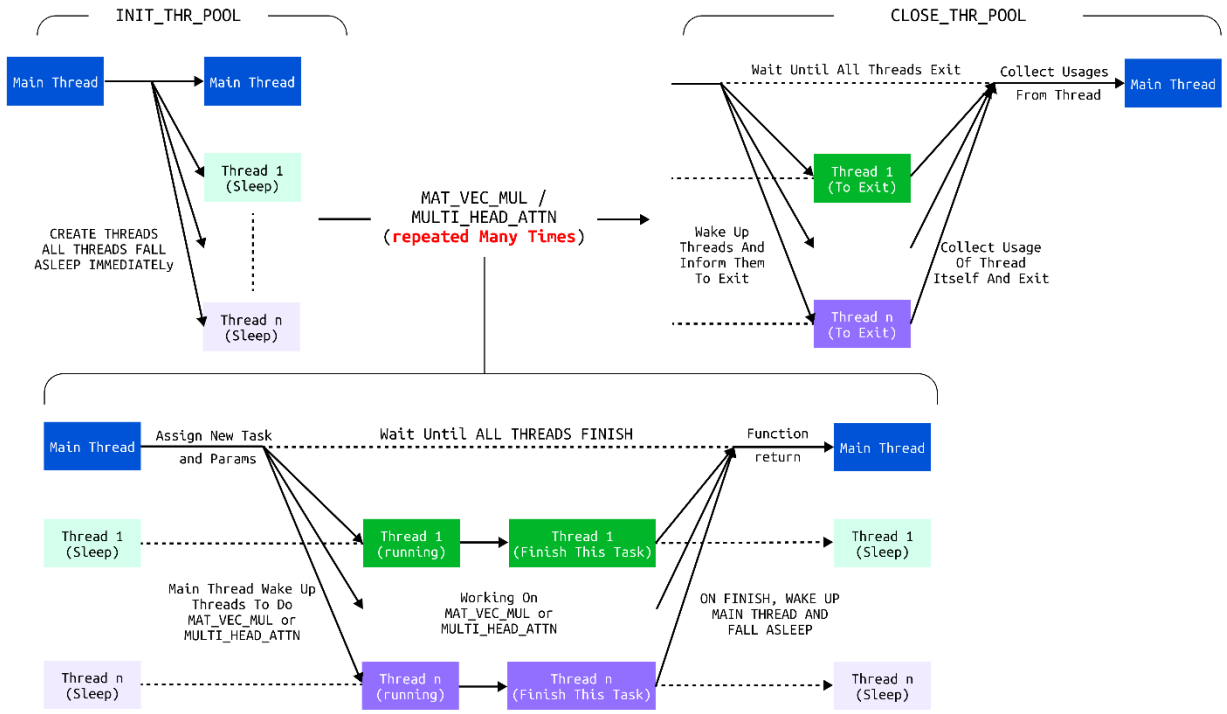

Figure 4. Reference Synchronization Workflow, consisting of 3 functions: a) INIT_THR_POOL function: create N threads, each thread falls asleep immediately; b) MAT_VEC_MUL or MULTI_HEAD_ATTN function: assign new parameters and tasks, wake up all threads to work, and wait until threads to finish before returned; c) CLOSE_THR_POOL function: wake up threads to collect system usage and exit, wait until all threads to exit and collect usage of terminated threads.

More specifically, the synchronization workflow in Figure 4 consists of 4 functions and a thread function:

1. `init_thr_pool(int thr_count)`: to be called by the main thread at the beginning of the program, shall:

    a. Create `thr_count` threads
    b. Let threads identify themselves, i.e., thread knows I am the i-th thread
    c. Let the created threads go to wait immediately

2. `void mat_vec_mul(float* out, float* vec, float* mat,...)`: API exposed to do Matrix-Vector-Multiplication, **signature must be same as sequential version**, shall:

    a. Assign the `mat_vec_mul` computation and the parameters (`out, vec, mat, ...`) to threads
    b. Wake up threads to do the calculation
    c. Main thread waits until all threads to complete the calculation

3. `void multi_head_attn(float* out, float* q, ...)`: API exposed to do Multi-Head Attention, **signature must be same as sequential version**, shall:

    d. Assign the `multi_head_attn` computation and the parameters (`out, q, ...`) to threads
    e. Wake up threads to do the calculation
    f. Main thread waits until all threads to complete the calculation

4. `close_thr_pool()`: to be called at the end of the program, shall:

    a. Wake up all threads and inform them to collect their system usage and terminate

b. Wait until all threads exit, and collect and print the system usage of all terminated threads
c. Collect and print the system usage of the main thread as well as the whole program
d. Release all resources related to multi-threading

5. `void* thr_func(void* arg)`: thread function, shall:

   a. Immediately wait for synchronization after initialization
   b. Can be woken up by the main thread to work on the assigned computation (i.e., based on the task and parameters)
   c. After finishing the current workload, inform the main thread and go back to wait
   d. Being able to terminate and collect its system usage

It's worth noticing that you should create one thread pool that can properly handle both `mat_vec_mul` and `multi_head_attn` instead of two pools (one for each) Thus, you may also need to implement the following:

6. `mat_vec_mul_task_func(int id, ...)`: function executed by each thread to perform (its portion of) matrix-vector-multiplication. The first argument is the thread id in the pool (not tid in OS), and the rest of the signature is left for you to design and implement.
7. `multi_head_attn_task_func(int id, ...)`: function executed by each thread to perform (its portion of) multi-head attention. The first argument is the thread id in the pool (not tid in OS), and the rest of the signature is left for you to design and implement.

More details and reasons behind the design can be found in Appendix b. Context Design.

There might be other synchronization workflows, and we are open to your ideas. However, due to the large class size, we can only accept submissions following the above design.

Specifications

a. Preparing Environment

Download start code – Download `start.zip` from the course's Moodle, and unzip to a folder with:

```
start
├── common.h          # common and helper macro definitions, read through first
├── seq.c             # start point, single-thread inference engine
├── parallel_[UID].c  # [your task] template including synchronization functions above
├── Makefile          # makefile for the project, update [UID] on line 5
└── model.h           # GPT model definition, modification not allowed
```

Download the model files. There are two files required, model.bin for model weight and tokenizer.bin for tokenizer. Please use the following instructions to download them:

```
make prepare # will download if not existed
# or manually download via wget, will force repeated download, not recommended
wget -O model.bin https://huggingface.co/huangs0/smollm/resolve/main/model.bin
wget -O tokenizer.bin https://huggingface.co/huangs0/smollm/resolve/main/tokenizer.bin
```

Compile and run the inference program. The **initial** `seq.c` is a complete single-thread (sequential) C inference program that can be compiled as follows:

```
make -B seq # -B := --always-make, force rebuild
gcc -o seq seq.c -O2 -lm # or manually make with gcc
```

Please use `-lm` flag to link the Math library and `-O2` flag to apply level-2 optimization. Please use `-O2` and don't use other optimization for fairness.

Run the compiled seq program. The program can be executed with an integer specifying the random seed and a quoted string specifying the prompt. For simplicity, we fixed it to a single prompt.

```
./seq <seed> <prompt> # prompt must quoted with "", recommend using only one prompt
# examples, more to be found in bench.txt
./seq 42 "What's Fibonacci Number?"
./seq 42 "Why didn't my parents invite me to their wedding?"
```

Upon invocation, the program will configure the random seed and begin sentence generation based on the provided prompt. The program calls the `forward` function to call LLM and generate the next token, and the `printf()` with `fflush()` to print the generated word to the terminal immediately. A pair of utility time measurement functions `time_in_ms` will measure the time with millisecond accuracy.
Finally, when generation is finished, the length, average speed, and system usage will be printed:

```
$ ./seq 42 "What is Fibonacci Number?"
User: What is Fibonacci Number?
assistant
A Fibonacci sequence is a sequence of numbers in which each number is the sum of the two
preceding numbers (1, 1, 2, 3, 5, 8, 13, ...)
......
F(n) = F(n-1) + F(n-2) where F(n) is the nth Fibonacci number.
length: 266, speed (tok/s): 21.4759
main thread – user: 12.4495 s, system: 0.0240 s
```

By **fixing the same machine (workbench2) and the same random seed, generated text can be exactly replicated**. For example, the above sample is from the test we conducted on workbench2 with random seed 42. Moreover, `achieved tok/s` represents the average number of tokens generated within a second, and **we use it as the metric for speed measurement**. Due to the fluctuating system load from time to time, the speed of the generation will **fluctuate around some level**.

b. Implement the parallel mat_vec_mul and multi_head_attn by multi-threading

Open the `parallel_[UID].c`, rename `[UID]` with your UID, and open Makefile, rename [UID] with your UID (make sure no blank space after). Implement the thread pool with the workflow illustrated in Figure. 4 by completing the required five functions and adding appropriate global variables or more functions if needed.

For multi-threading, please use **pthread.h**. For synchronization, please use **either semaphore or (mutex locks and conditional variables)**. You can only modify the code between specified `// YOUR CODE STARTS HERE` at line 28 and `// YOUR CODE ENDS HERE` at line 88.

Here are some suggestions for the implementation:

1. How to assign new parameters to threads and inform threads to terminate?
   i. Via variables in global space. Noted that thr_func can access global variables.
   ii. Via pointers to parameters. The main thread changes pointers to params before wake-up worker threads.
2. How to assign new tasks (mat_vec_mul or multi_head_attn)? Via function pointers.
3. Once the main thread invokes mat_vec_mul() or multi_head_attn(), it should wait for all computations to complete before returning from the function.

4. For collecting system usage upon finish, please use `getrusage`.

Your implementation shall be able to be compiled by the following command:

```
make -B # applicable after renaming [UID]
gcc -o parallel parallel_[UID].c -O2 -lm -lpthread # or manually via gcc
```

The program accepts three arguments, (1) `thr_count`, (2) `seed`, and (3) the prompt (enclosed by ""). Code related to reading arguments has been provided in `parallel_[UID].c`. You can use `thr_count` to specify the number of threads to use.

```
./parallel <thr_count> <seed> <prompt>
```

If your implementation is correct, **under the same random seed, the generated text shall be the same as the sequential version, but the generation will be faster**. Moreover, you should report on the system usage for each thread respectively (including the main thread) and the whole program. For example, this is the output of random seed 42 on workbench2 with 4 threads:

```
./parallel 4 42 "What is Fibonacci Number?"
User: What is Fibonacci Number?
assistant
A Fibonacci sequence is a sequence of numbers in which each number is the sum of the two
preceding numbers (1, 1, 2, 3, 5, 8, 13, ...)
......
F(n) = F(n-1) + F(n-2) where F(n) is the nth Fibonacci number.
length: 266, speed (tok/s): 38.8889
Thread 0 has completed - user: 4.9396 s, system: 0.1620 s
Thread 1 has completed - user: 4.7195 s, system: 0.1806 s
Thread 2 has completed - user: 4.6274 s, system: 0.1843 s
Thread 3 has completed - user: 5.0763 s, system: 0.1702 s
main thread - user: 0.6361 s, system: 0.6993 s
Whole process - user: 20.0198 s, system: 1.3757 s
```

c. Measure the performance and report your findings

Benchmark your implementation (tok/s) **on your environment** (Linux or WSL or docker or Github Codespaces) with different thread numbers and report metrics like the following table:

Thread Numbers	Speed (tok/s)	User Time	System Time	Use Time/System Time
0 (**Sequential**)				
1				
2				
4				
6				
8				
10				
12				
16				

Regarding system usage (user time / system time), please report the usage of the whole process instead of each thread. Then based on the above table, try to briefly analyze the relation between performance and the number of threads and reason the relationship. Submit the table, your analysis, and your reasoning in a **one-page pdf** document.

IMPORTANT: Due to the large number of students this year, **please conduct the benchmark on your own computer instead of the workbench2 server**. The grading of your report is based on your analysis and reasoning instead of the speed you achieved. When you're working on workbench2, please be reminded that you have limited maximum allowed thread numbers (128) and process (512), so please do not conduct benchmarking on the workbench2 server.

Submission

Submit your program to the Programming # 2 submission page on the course's Moodle website. Name the program to `parallel _[UID].c` (replace `[UID]` with your HKU student number). As the Moodle site may not accept source code submission, you can compress files to the zip format before uploading. Submission checklist:

- Your source code `parallel_[UID].c`, must be self-contained with no dependencies other than `model.h` and `common.h`) and `Makefile`.
- Your report must include the benchmark table, your analysis, and your reasoning.
- Your GenAI usage report contains GenAI models used (if any), prompts and responses.
- Please **do not** compress and submit the model and tokenizer binary files. (`make clean_bin`)

Documentation

1. At the head of the submitted source code, state the:

    - File name
    - Student's Name and UID
    - Development Platform (Please include compiler version by `gcc -v`)
    - Remark – describe how much you have completed (See Grading Criteria)

2. Inline comments (try to be detailed so that your code can be understood by others easily)

Computer Platform to Use

For this assignment, you can develop and test your program on any Linux platform, but **you must make sure that the program can correctly execute on the workbench2 Linux server** (as the tutors will use this platform to do the grading). Your program must be written in C and successfully compiled with gcc on the server.

It's worth noticing that the **only server for COMP3230 is workbench2.cs.hku.hk**, and **please do not use any CS department server, especially academy11 and academy21**, as they are reserved for other courses. In case you cannot login to workbench2, please contact the tutor(s) for help.

Grading Criteria

1. Your submission will be primarily tested on the workbench2 server. Make sure that **your program can be compiled without any errors**. Otherwise, we have no way to test your submission, and you will get a zero mark.
2. As the tutor will check your source code, please write your program with good readability (i.e., with clear code and sufficient comments) so that you will not lose marks due to confusion.
3. You can only use pthread.h and semaphore.h (if needed), **using other external libraries like OpenMP, BLAS or LAPACK will lead to a 0 mark**.

Detailed Grading Criteria

- Documentation **-1 point if failed to do**
  - Include necessary documentation to explain the logic of the program
  - Include the required student's info at the beginning of the program
- Report: **1 point**
  - Measure the performance of the sequential program and your parallel program **on your computer** with various No. of threads (0, 1, 2, 4, 6, 8, 10, 12, 16).
  - Briefly analyze the relation between performance and No. of threads, and reason the relation
- Implementation: **11 points evaluated progressively**

1. (+3 points = 3 points) **Achieve correct result & use multi-threading**. Correct means generated text of multi-threading and sequential are identical with the same random seed.
2. (+3 points = 6 points total) **All in 1., and achieve >10% acceleration by multi-threading compared with sequential under 4 threads**. Acceleration measurement is based on tok/s, acceleration must result from multi-threading instead of others like compiler (`-O3`), etc.
3. (+2 points = 8 points total) **All in 2., and reuse threads in multi-threading**. Reuse threads means the number of threads created in the whole program must be constant as thr_count.
4. (+3 points = 11 points total) **All in 3., and mat_vec_mul and multi_head_attn use the same thread pool**. Reusing the same thread pool means there's only one pool and one thread group.

Plagiarism

Plagiarism is a very serious offense. Students should understand what constitutes plagiarism, the consequences of committing an offense of plagiarism, and how to avoid it. **Please note that we may request you to explain to us how your program is functioning as well as we may also make use of software tools to detect software plagiarism.**

GenAI Usage Guide

Following course syllabus, you are allowed to use Generative AI to help completing the assignment, and please clearly state the GenAI usage in GenAI Report, including:

- Which GenAI models you used
- Your conversations, including your prompts and the responses

Appendix

a. Parallelism Checking

To parallel by multi-threading, it's critical to verify if the computation is independent to avoid race conditions and the potential use of lock. More specifically, we need to pay special attention to check and avoid writing to the same memory location while persisting the correctness.

For example, the 1st iteration (outer for-loop) matches the requirement of independence as the computation of each row won't affect others, and the only two writing is `out[i]` and `val`. Writing to the same `out[i]` can be avoided by separating `i` between threads. `val` can be implemented as stack variables for each thread respectively so no writing to the same memory.

Quite the opposite, 2nd iteration (inner for-loop) is not a good example for multi-threading, though the only writing is `val`. If `val` is implemented as a stack variable, then each thread only holds a part of the correct answer. If `val` is implemented as heap variables to be shared among threads, then `val` requires a lock for every writing to avoid race writing, which is obviously costly.

b. Design of Context

A straightforward solution to the above problem is to let `thr_func` (thread function) do computation and exit when finished and let original mat_vec_mul or `multi_head_attn` to create threads and wait for threads to exit by `pthread_join`. This could provide the same synchronization.

However, this implementation is problematic because each function call to mat_vec_mul or `multi_head_attn` will create n new threads. Unfortunately, to generate a sentence, GPTs will call mat_vec_mul or `multi_head_attn` thousands of times, so thousands of threads will be created and destroyed, which leads to significant overhead to the operation system.

Noted that all the calls to mat_vec_mul function or `multi_head_attn` are doing the same task, i.e., Matrix-Vector-Multiplication, and the only difference between each function call is the parameters. Thus, a straightforward optimization is to **reuse the threads**. In high-level, we can create *N* threads in advance, and when mat_vec_mul or `multi_head_attn` is called, we assign new parameters for thread functions and let threads work on new parameters.

Moreover, it's worth noting that **mat_vec_mul and `multi_head_attn` are only valid within the context,** i.e., between `init_thr_pool` and `close_thr_pool`, or there are no threads other than the main (not yet created or has been destroyed). This kind of context provides efficient and robust control over local variables and has been integrated with high-level languages like Python.