COMP3230 Principles of Operating Systems

Programming Assignment Two

Due date: Nov. 19, 2023, at 23:59

Total 11 points

Programming Exercise – Accelerate Large Language Model Inference

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 direct practice in designing and developing multithreading 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

Optimize the matrix-vector-multiplication algorithm of GPT by multi-threading. Similar to other neural networks, GPT and its variations utilize matrix-vector-multiplication, or called fully-connected/linear layer in DL, to apply the parameter learned, which takes >70% of the whole calculation. Thus, to accelerate the GPT and get faster response, it's critical to have faster matrix-vector-multiplication, and multi-threading are usually considered powerful.

In this assignment, we will use an open-source variation of GPT, llama2 released by Meta, and we provide a complete pure C implementation of its inference in **`seq.c`** as the baseline of your work, along with model weights. You need to use `pthread.h` with either the semaphore or (mutex_lock + conditional variable) to implement a multi-threading version of matrix-vector-multiplication. This multi-threading version will significantly accelerate the inference of Large Language Model.

Acknowledgement: This assignment is based on the open-source project llama2.c by Andrej Karpathy, thanks open-source.

GPT-based Large Language Model

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

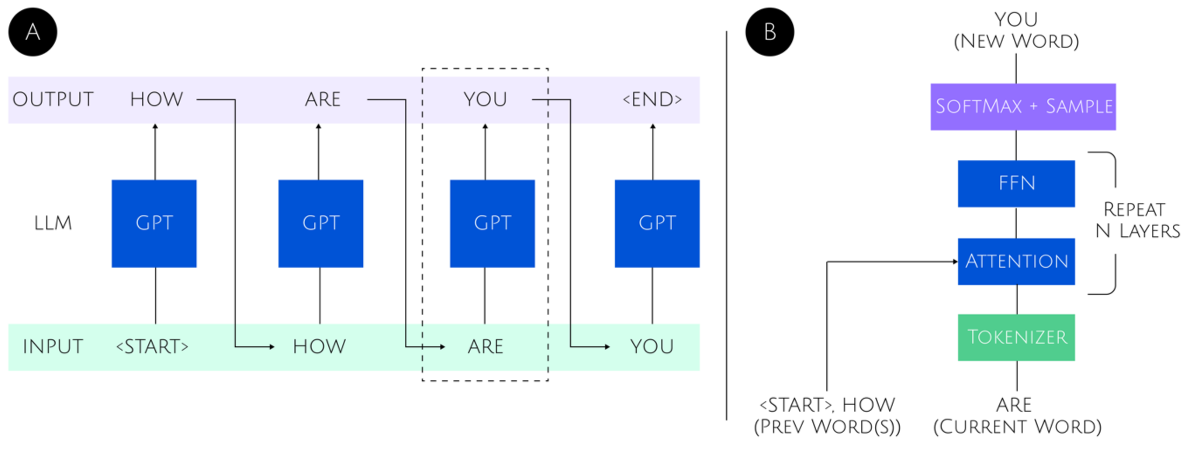


Figure 1. GPT Insight. a) GPT generate text one by one, and each output is the input of next generation. b) GPT has four major components: Tokenizer turns word (string) into vector, Softmax + Sample give next token, and each layer has Attention and FFN (Feed-Forward Network), consisting of many Matrix-Vector-Multiplication

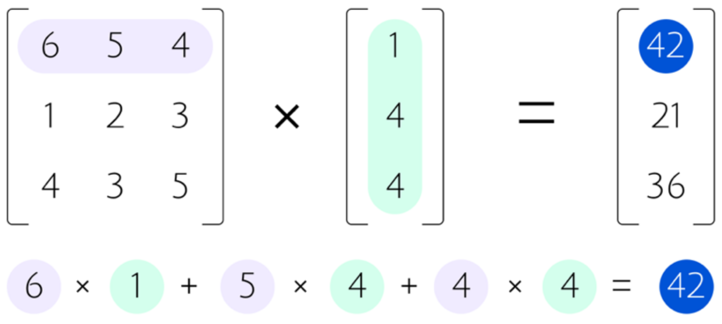
Figure 1b showcases the inference workflow of each word like “You” in “How are you”: First, words are transformed into tokens using a tokenizer, which is essentially a (python) dictionary that assigns a unique vector to each word. The embedding vectors go through multiple layers, each consisting of three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

- The first step is attention, where the model calculates attention scores based on the cosine similarity between the current word's query embedding and the embeddings of previous words (keys). The attention output is a weighted average of the value embeddings, and this process involves learnable parameters in the form of Matrix-Vector-Multiplication (linear layer).
- The second step is a feed-forward network (FFN) that adds 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 to the attention calculations.

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 samples a word from the distribution.

Understanding GPT is not required for this assignment. Just remember that LLM uses a lot of Matrix-Vector-Multiplication to apply learned parameters to make it powerful.

Task: Matrix-Vector-Multiplication



$$\begin{bmatrix} 6 & 5 & 4 \\ 1 & 2 & 3 \\ 4 & 3 & 5 \end{bmatrix} \times \begin{bmatrix} 1 \\ 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 42 \\ 21 \\ 36 \end{bmatrix}$$

$$6 \times 1 + 5 \times 4 + 4 \times 4 = 42$$

Figure 2. Matrix-Vector-Multiplication Algorithm.

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

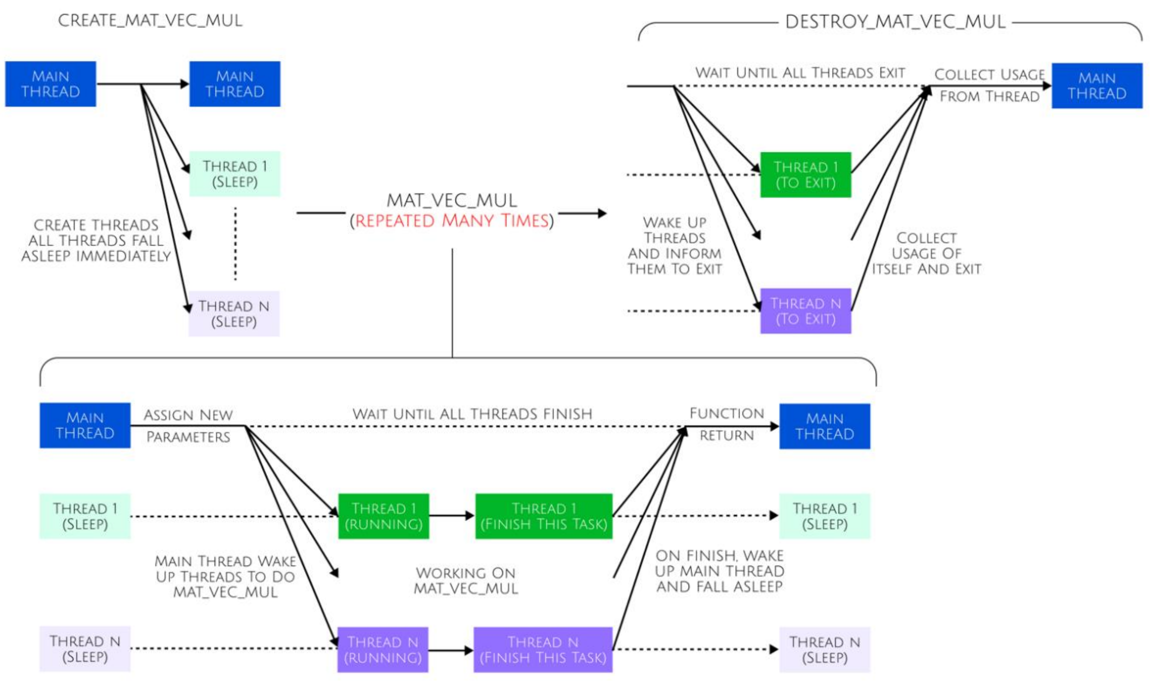
```
For Each Row i
    For Column j, accumulate Matrix[i][j] * Vector[j] to Out[i]
```

More specifically, a sample C implementation is shown below (also in seq.c):

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

Your task in this assignment is to parallelize the outer iteration (at the 2nd line) by allocating rows to threads. More specifically, in the case of a Matrix with d rows and n threads working on the computation, if d is divisible by n, the k-th thread ($k = 0, 1, \dots, n-1$) will handle the rows from $\left[k \times {}^{d}/_{n}\right]$ to $\left[(k+1) \times {}^{d}/_{n} - 1\right]$. 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 first $n-1$ threads ($k = 0, 1, \dots, n-2$) with $\left\lceil {}^{d}/_{n} \right\rceil$ rows, while the last thread handles remaining rows. More explanation on such design can be found on Appendix a. Parallel Checking.

Moreover, in order to reduce overhead, you are required to create one set of threads and reuse them for all `mat_vec_mul()` function calls, instead of creating threads for each `mat_vec_mul()` function call. One popular way based on Synchronization is illustrated in Figure 3.



CREATE_MAT_VEC_MUL
MAIN THREAD
MAIN THREAD
THREAD 1 (SLEEP)
CREATE THREADS ALL THREADS FALL ASLEEP IMMEDIATELY
THREAD N (SLEEP)
MAT_VEC_MUL (REPEATED MANY TIMES)
DESTROY_MAT_VEC_MUL
WAIT UNTIL ALL THREADS EXIT
COLLECT USAGE FROM THREAD
MAIN THREAD
THREAD 1 (TO EXIT)
WAKE UP THREADS AND INFORM THEM TO EXIT
COLLECT USAGE OF ITSELF AND EXIT
THREAD N (TO EXIT)
MAIN THREAD
ASSIGN NEW PARAMETERS
WAIT UNTIL ALL THREADS FINISH
FUNCTION RETURN
MAIN THREAD
THREAD 1 (SLEEP)
THREAD 1 (RUNNING)
THREAD 1 (FINISH THIS TASK)
THREAD 1 (SLEEP)
MAIN THREAD WAKE UP THREADS TO DO MAT_VEC_MUL
WORKING ON MAT_VEC_MUL
ON FINISH, WAKE UP MAIN THREAD AND FALL ASLEEP
THREAD N (SLEEP)
THREAD N (RUNNING)
THREAD N (FINISH THIS TASK)
THREAD N (SLEEP)

Figure 3. Reference Synchronization Workflow, consisting of 3 function: a) CREATE_MAT_VEC_MUL function: create n threads, each threads fall asleep immediately; b) MAT_VEC_MUL function: assign new parameters, wake up threads to work on parameters and wait until threads to finish to return; c) DESTROY_MAT_VEC_MUL function: wake up threads to collect system usage and exit, wait until threads to exit and collect usage of threads.

More specifically, the synchronization workflow illustrated in Figure 3 consists of 3 functions and the thread function:

1. `create_mat_vec_mul(int thr_count)`: to be called at the beginning of program, shall:

    a. Create n threads
    b. Let threads identify themselves, i.e., thread knows I am the i-th threads
    c. Let the created threads fall asleep immediately

2. `void mat_vec_mul(float* out, float* vec, float* mat, int col, int row)`: API exposed to do Matrix-Vector-Multiplication, shall:

    a. Assign new parameters (`out, vec, mat, col, row`) to threads
    b. Wake up threads to do calculation
    c. Main thread waits until threads to complete

3. `destroy_mat_vec_mul()`: to be called at the end of program, shall:

    a. Wake up threads to collect the system usage (of themselves) and terminates
    b. Wait until all threads to exit and collect system usage
    c. Clear all other resources related with multi-threading

4. `void* thr_func(void* arg)`: Thread function to do Matrix-Vector-Multiplication, shall:

    a. Fall asleep immediately after initialization
    b. Can be woke up by main thread to work
    c. After finishing the current work, inform main thread
    d. Being able to terminate

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

Definitely there might have other synchronization workflow, and we are open to your ideas. However, due to the large class size, we can only accept submissions following the above design structure.

Specifications

a. Preparing Environment

Download the sequential program – `seq.c`, along with utility functions in `utilities.c` and `utilities.h` from course's Moodle site. Compile the program with *gcc*:

```
gcc -o seq seq.c utilities.c -O2 -lm
```

Please include utilities.c, use `-lm` flag to link math library and `-O2` flag to apply level-2 optimization. Please stick to `-O2` and don't use other optimization for fairness. You don't need to understand and are not allowed to modify `utilities.c` and `utilities.h`.

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

```
wget -O model.bin https://huggingface.co/huangs0/llama2.c/resolve/main/model.bin
wget -O tokenizer.bin https://huggingface.co/huangs0/llama2.c/resolve/main/tokenizer.bin
```

Run the compiled program by giving an integer as the random seed for sampling.

```
./seq <seed>
```

Upon invocation, the program will configure the random seed and begin sentence generation starting from a special `<START>` token. The program call `transformer` function to generate the next token, and `printf` with `fflush` to print the generated word to shell immediately. A pair of utility time measurement function `time_in_ms` will measure the time in millisecond accuracy:

```c
long start = time_in_ms(); // measure time in ms accuracy
int next, token = 1, pos = 0; // token = 1 -> <START>

while (pos < config.seq_len) { // not exceed max length
    next = transformer(token, pos, &config, &state, &weights); // generate next
    printf("%s", vocab[next]); fflush(stdout); // force print
    token = next; pos++; // record token and shift position
}

long end = time_in_ms(); // measure time in ms accuracy
```

This program will start generating tiny stories. Finally, when generation is finished, the length of the generated text, total time, average speed, and system usage will be printed such as:

```
One day, a little girl named Lucy
......
Carrying a brightly stepped for one dog ladybuging once she had

length: 256, time: 4.400000 s, achieved tok/s: 58.181818
main thread - user: 4.3881 s, system: 0.0599 s
```

By **fixing the same machine (workbench2) and the same random seed, generated text can be exactly replicated**. For example, the above sample is 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 Matrix-Vector-Multiplication by multi-threading

Open the `llama2_[UID].c`, rename `[UID]` with your UID, and implement the workflow illustrated in Figure. 3 by completing the four functions and adding appropriate global variables. 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 38 and `// YOUR CODE ENDS HERE` at line 66 in `llama2_[UID].c`.

Here are some suggestions for the implementation:

1. How to assign new tasks and inform them to terminate? Noted that all threads can access global variables so you can update the global variables and wake them up.
2. Main thread shall wait for threads to work or terminate.
3. For collecting system usage, please consider `getrusage`.

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

```
gcc -o llama2_[UID] llama2_[UID].c utilities.c -O2 -pthread -lm
```

Then run the compiled program. Now it accepts two arguments `seed` and `thr_count`. Code related to reading arguments has been provided in `llama2_[UID].c`. You can use `thr_count` to specify the number of threads to use.

```
./llama2_[UID] <seed> <thr_count>
```

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

```
One day, a little girl named Lucy
......
Carrying a brightly stepped for one dog ladybuging once she had

length: 256, time: 2.100000 s, achieved tok/s: 121.904762
Thread 0 has completed - user: 1.2769 s, system: 0.0363 s
Thread 1 has completed - user: 1.2658 s, system: 0.0361 s
Thread 2 has completed - user: 1.2749 s, system: 0.0277 s
Thread 3 has completed - user: 1.2663 s, system: 0.0323 s
main thread - user: 5.7126 s, system: 0.3919 s
```

c. Measure the performance and report your finding

Benchmark your implementation (tok/s) **on your own computer** 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 above table, try to briefly analyze relation between performance and No. threads and reason the relationship. Submit the table, your analysis and 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**. 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 workbench2 server.

Submission

Submit your program to the Programming # 2 submission page at the course's Moodle website. Name the program to `llama2_[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 `llama2_[UID].c`, must be self-contained. (No dependencies other than `utilities.c` and `utilities.h`)
- Your report including benchmark table, your analysis and reasoning
- Please **do not** compress and submit model and tokenizer binary file

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 could be understood by others easily)

Computer Platform to Use

For this assignment, you can develop and test your program on any Linux/WSL/docker 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.

Please note 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 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 good code convention and sufficient comments) so that you will not lose marks due to confusion.
3. You can only use pthread.h and semaphore.h(if need), using other external libraries like OpenMP, LAPACK will lead to 0 mark.

Detailed Grading Criteria

- Documentation **-1 point if failed to do**
  - Include necessary documentation to explain the logic of the program
  - Include 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. threads (0, 1, 2, 4, 6, 8, 10, 12, 16).
  - Briefly analyze the relation between performance and No. threads and reason the relation
- Implementation: **10 points evaluated progressively**

  1. (+2 points = 2 points) **Achieve correct result & use multi-threading**. Correct means generated text of multi-threading and sequential are identical with same random seed.
  2. (+3 points = 5 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. (+5 points = 10 points total) **All in 2., and reuse threads in multi-threading**. Reuse threads means number of threads created in the whole program must be constant as thr_count.

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.**

Note: You must clearly acknowledge it if you use ChatGPT or any AI tools to generate code in your implementation.

Appendix

a. Parallelism Checking

To parallel by multi-threading, it’s critical to verify if the computation is independent to avoid race condition 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, 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 avoid 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 stack variable, then each thread only holds a part of correct answer. If val is implemented as heap variables to be shared among threads, then val requires lock to avoid race writing.

b. Design of Context

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

However, this implementation is problematic because each function call to mat_vec_mul will create n new threads. Unfortunately, to generate a sentence, LLM like llama2 will call mat_vec_mul thousands of times, so thousands of threads will be created and destroyed, which leads to indefinite overhead to the operation system.

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

Moreover, It’s worth noticed that mat_vec_mul is only valid within the context, i.e., between create_mat_vec_mul and destroy_mat_vec_mul, 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 variable, and has been integrated with high-level languages like Python `with`.