6.5940 Fall 2024 Lab 5: Optimize LLM on Edge Devices

Local chatbot on Macbook M1 Pro

Running large language models (LLMs) on the edge is of great importance. By embedding LLMs directly into real-world systems such as in-car entertainment systems or spaceship control interfaces, users can access instant responses and services without relying on a stable internet connection. Moreover, this approach alleviates the inconvenience of queuing delays often associated with cloud services. As such, running LLMs on the edge not only enhances user experience but also addresses privacy concerns, as sensitive data remains localized and reduces the risk of potential breaches.

However, despite their impressive capabilities, LLMs have traditionally been quite resource-intensive. They require considerable computational power and memory resources, which makes it challenging to run these models on edge devices with limited capabilities.

In this lab, you will learn the following:

- How to deploy an LLaMA2-7B-chat with TinyChatEngine on your computer.
- Implement different optimization techniques (loop unrolling, multithreading, and SIMD programming) for the linear kernel.
- Observe the end-to-end latency improvement achieved by each technique.

Prerequisites:

- Basic C and C++ programming (Online resource for C/C++ introduction: 6.S096 Effective Programming In C And C++)
- It's recommended to read this parallel computing tutorial which introduces optimization techniques.

System requirement

In this lab, we will deploy a quantized LLM model on your computer to run a chatbot. The minimum requirement is as follows:

- Operating systems: MacOS, Linux, or Windows
- Processor: x86 (Intal/AMD) or ARM (Apple M1/M2)
- Memory: 8 GB
- Available storage: 5 GB

It is highly recommended to prepare a personal computer/laptop to complete this assignment. However, if your computer cannot meet the above requirement, you may use the following computing resources provided by MIT:

1. Athena Computing Environment
    - Link: https://ist.mit.edu/athena
2. Computers at MIT Libraries
    - Map: https://ist.mit.edu/sites/default/files/getting_started/ComputingMap.pdf
    - We've tested TinyChatEngine on the computers at Barker Engineering Library. However, they are not user-friendly and thus NOT recommended. Please contact TAs for more information if you would like to use MIT library computers.

Setup

Please follow the instructions for the setup on your computer.

- On MacOS: Install boost and llvm with Homebrew by the following commands using the terminal.

```
brew install boost
brew install llvm
```

  Make sure Python is available on your computer. If not, it is recommended to install it with Anaconda.

- On Linux-based operating systems (e.g., Ubuntu): Make sure Python is available on your computer. If not, it is recommended to install it with Anaconda.
- On Windows: Make sure the following commands are available on your computer.
    - g++
    - make
    - unzip
    - git

- Python

If not, you can download Python from the official website and install the other dependency with MSYS2. Please follow this tutorial for installation. Then install the required dependencies with the following:

```
pacman -S --needed base-devel mingw-w64-x86_64-toolchain make
unzip git
```

Download TinyChatEngine and LLaMA2-7B-chat model

Let's download TinyChatEngine and the LLaMA2-7B-chat model on your computer. This model will later be quantized and deployed to run a local chatbot.

- First, download the TinyChatEngine codebase.

```
git clone --recursive https://github.com/mit-han-lab/tinychat-tutorial
```

- Download the LLaMA2-7B-chat model from our model zoo. The model will be stored at `INT4/models/LLaMA_7B_2_chat/`. This download process will take a while…
  - For x86 devices (Intel/AMD):

```
cd <path-to-repo>/transformer
python download_model.py --model LLaMA_7B_2_chat --QM QM_x86
```

  - For ARM devices (M1/M2):

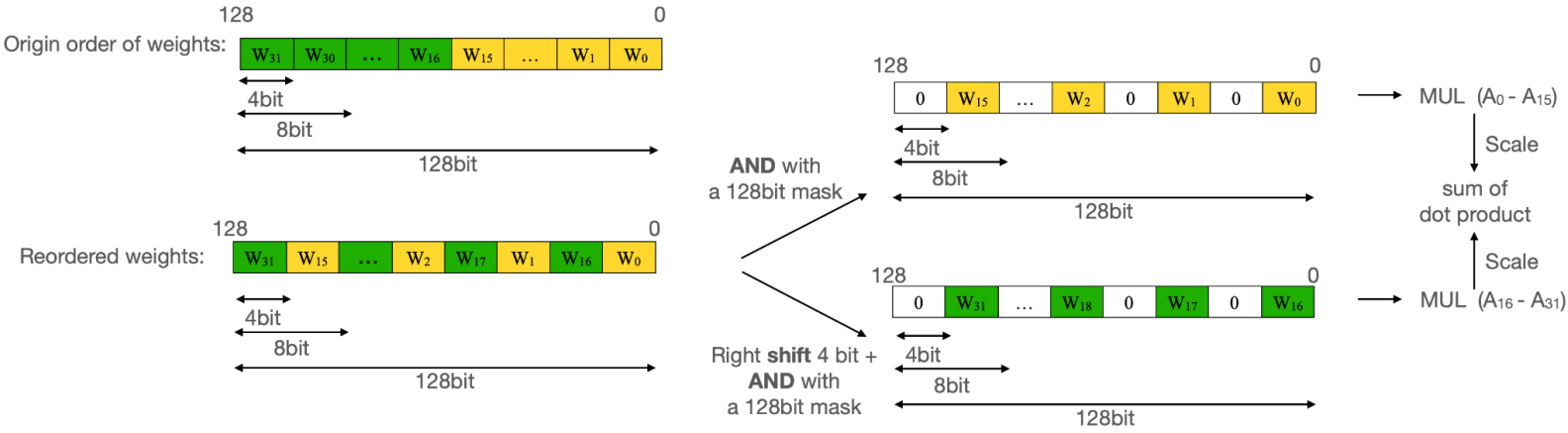
```
cd <path-to-repo>/transformer
python download_model.py --model LLaMA_7B_2_chat --QM QM_ARM
```

Background: Device-specific 4bit Weight Reordering

To mitigate the runtime overheads associated with weight reordering, TinyChatEngine conducts this process offline during model conversion. In this section, we will explore the weight layouts of QM_ARM and QM_x86. These layouts are tailored for ARM and x86 CPUs, supporting 128-bit SIMD and 256-bit SIMD operations, respectively.

- **Layout of QM_ARM:**

For QM_ARM, consider the initial configuration of a 128-bit weight vector, [w^0, w^1, ... , w^{30}, w^{31}], where each w^i is a 4-bit quantized weight. TinyChatEngine rearranges these weights in the sequence [w^0, w^{16}, w^1, w^{17}, ..., w^{15}, w^{31}] by interleaving the lower half and upper half of the weights. This new arrangement facilitates the decoding of both the lower and upper halves using 128-bit *AND* and *shift* operations, as depicted in the subsequent figure. This will eliminate runtime reordering overheads and improve performance.



- **Layout of QM_x86:**

Similarly, for QM_x86, TinyChatEngine rearranges a 256-bit weight vector, [w^0, w^1, ... , w^{62}, w^{63}], where each w^i is a 4-bit quantized weight into the sequence of [w^0, w^{32}, w^1, w^{33}, ..., w^{31}, w^{63}]. Such a layout ensures optimal compatibility with the 256-bit AVX2 SIMD instructions for efficient dequantization.

Deploy TinyChatEngine and Optimize the Linear Kernel

TinyChatEngine is a powerful neural network library specifically designed for the efficient deployment of quantized large language models (LLMs) on edge devices.

TinyChatEngine adopts the following optimization techniques to accelerate inference and minimize memory footprint.

- Loop unrolling: A loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size, which is an approach known as space-time tradeoff. (Lecture TinyEngine: Loop optimization)
- Multithreading: Multithreading is a programming concept that allows a single process to manage multiple threads of execution concurrently. Each thread shares the process's resources but can execute independently, leading to improved performance, smoother user experiences, and better resource utilization. (Lecture TinyEngine: Multithreading)
- SIMD (Single instruction, multiple data) instructions: A computing method that enables the processing of multiple data with a single instruction. (Lecture TinyEngine: SIMD programming)

In this lab, you will gain hands-on experience in implementing various optimization techniques for the w4a8 linear kernel, where the group size for quantization is **32**. By filling in the provided starter codes, you will enhance the performance of a given

algorithm. You can also evaluate the correctness and speedup achieved by the implementations with the provided script.

Overview of starter codes:

- File Structure: All starter codes are located inside the `kernels/starter_code` directory. Here's a breakdown:

```
kernels/
│     ...
└────starter_code/
    ├── reference.cc
    ├── loop_unrolling.cc
    ├── multithreading.cc
    ├── simd_programming.cc
    ├── multithreading_loop_unrolling.cc
    └── all_techniques.cc
transformer/
└──── evaluate.sh
```

- Reference Implementation: We provide a basic reference implementation (reference.cc) using a standard for loop. This will serve as the baseline for performance comparison.
- Optimization Techniques: You will fill in code and complete the other implementation, including:
    - Loop Unrolling (loop_unrolling.cc)
    - Multithreading (multithreading.cc)
    - SIMD Programming (simd_programming.cc)
    - Multithreading with Loop Unrolling (multithreading_loop_unrolling.cc)
    - Combination of All Techniques (all_techniques.cc)
- Evaluation Script: The provided bash script, `evaluate.sh`, will be used to test the correctness and measure the speedup of the students' implementations. The script can test individual optimization techniques or all techniques at once.

How to evaluate each implementation:

Use the `evaluate.sh` script to test your implementation. The script will compile and run your code and then provide feedback on the correctness and performance (GOPs) achieved.

- To test all implementations: `./evaluate.sh`.
- You can specify which optimization technique to test or test all techniques. For example: To test a specific implementation, e.g., the loop unrolling technique: `./evaluate.sh loop_unrolling.` This will also generate an executable named `chat`, which you can run to deploy a local chatbot with the specific

implementation. To play with the local chatbot: `./chat`
The supported argument:
    - reference
    - loop_unrolling
    - multithreading
    - simd_programming
    - multithreading_loop_unrolling
    - all_techniques
- Example output of `./evaluate.sh reference`

```
-------- Sanity check of reference implementation: Passed! -------
Section, Total time(ms), Average time(ms), Count, GOPs
reference, 1511.364990, 15.113000, 100, 17.344850
```

- Evaluation: The script will compile and run your code, comparing it against the reference implementation. It will provide feedback on the correctness and performance (GOPs) achieved.

Note: Before diving into implementing each technique, it's recommended to ensure you have the necessary dependencies installed to compile and run the program by running `./evaluate.sh reference`

Instructions to implement each technique:

1. Complete the provided starter code by adding the necessary code segments, which are marked with "TODO" in the comments. You **only** need to write the code for your device, i.e., QM_x86 for x86 CPUs and QM_ARM for ARM CPUs. It's recommended to consult the reference implementation (***reference.cc***) and work on the templates in the following sequence:
    a. Loop Unrolling (loop_unrolling.cc)
    b. Multithreading (multithreading.cc)
    c. SIMD Programming (simd_programming.cc)
    d. Multithreading with Loop Unrolling (multithreading_loop_unrolling.cc)
    e. Combination of All Techniques (all_techniques.cc)
2. Evaluate the correctness of the implementation using the evaluation script.

Grading Policy:

- Total Points: 120 points (20 points for each optimization implementation and 20 points for bonus).
    - Correctness: 15 points. This will be based on the output of the evaluation script.
    - Performance Reporting: 5 points. Measure the performance improvement achieved by each technique on your computer and why it improves the performance.

    - Bonus: Any optimization techniques on your mind? Try to implement them to improve the performance further! If you can further improve the performance compared to the optimized kernel in TinyChatEngine, you can get bonus points here! Each percent of performance speedup equals one point up to 20 points.
- Submission:
    - Report: Please write a report (form) that includes your code and the performance improvement for each starter code.
        - Report template: https://docs.google.com/document/d/17Z_ab8EhDvjcigLXdDqMqd2LTVsZ4CnpOYNkRTrnTmU/edit?usp=sharing
    - Code: Use `git diff` to generate a patch for your implementation. We will use this patch to test the correctness of your code. Please name your patch as `{studentID}-{ISA}.patch` where {ISA} should be one of x86 and ARM, depending on your computer.