Spring 2025

CS 133 Lab 1

CPU w/ OpenMP: General Matrix Multiplication (GEMM)

Due Date: 4/17/25 2:00PM PST

Description

Your task is to parallelize general matrix multiplication (GEMM) with OpenMP based on the sequential implementation. Specifically, for matrix multiplication$C[I][J] = A[I][K] \times B[K][J]$, you will need to implement two functions using OpenMP:

```
void GemmParallel(const float a[kI][kK], const float b[kK][kJ], float c[kI][kJ]);
void GemmParallelBlocked(const float a[kI][kK], const float b[kK][kJ], float
c[kI][kJ]);
```

`GemmParallel`: A parallel version of the given matrix multiplication using OpenMP pragmas. Please note that some loop permutation might be needed to enable more concurrency.

`GemmParallelBlocked`: Blocked matrix multiplication with OpenMP. You should select the block size with the best performance when you submit this lab.

You are welcome to add any other optimization to further increase the performance. Please describe every optimization you performed in the report. You can assume the matrix sizes are multiples of 1024 (e.g. 1024x2048 (A) * 2048x4096 (B) = 1024x4096 (C)). You can change the matrix size in "`lib/gemm.h`". We may use different sizes other than 4096^3 in `GemmParallel`.

How-To

Build and Test Baseline GEMM

We have prepared the starter kit for you at BruinLearn/file/labs/lab1.zip. After you download and unzip the file, log in to your AWS instance or use the terminal of your own computer (recommended) to run the following commands:

```
cd lab1
./setup.sh        # install dependencies
make gemm
./gemm
```

It should run without error. Currently, we only provide support for Ubuntu and macOS. If you are running another operating system, please replace "`./setup.sh`" with the commands to install GCC on your operating system or install a Ubuntu virtual machine to run "`./setup.sh`". To specify a

special compiler for `make`, please use the environment variable `CXX` and/or `LDFLAGS`: e.g, `CXX=icpc make gemm`.

Create Your Own GEMM with OpenMP

If you have successfully built and run the baseline, you can start to create your own GEMM kernel. The provided starter kit will generate the test data and verify your results with a black-box implementation.

Your first task is to implement a parallel version of GEMM. You can start with the sequential version `GemmSequential` provided in `lib/gemm.cpp`. You should edit `omp.cpp` for this task.

Your second task is to implement a parallel version of blocked GEMM. You should continue working on your results from the first task. You should edit `omp-blocked.cpp` for this task.

To test your own implementation of GEMM:

```
make gemm
./gemm parallel                 # to test omp.cpp
./gemm parallel-blocked         # to test omp-blocked.cpp
```

If you see something similar to the following message, your implementation is incorrect.

```
Diff: 1.04936e+06
Your answer is INCORRECT!
```

Your code's performance will be shown after `Perf:` in GFLOPS.

Create an AWS Instance for Performance Experiments

Please refer to the discussion section slides and create an `m5.2xlarge` instance with Ubuntu 22.04 AMI.

Tips

- We will use `m5.2xlarge` instances for grading.
- If you develop on AWS:
    - To resume a session in case you lose your connection, you can run `screen` after login. You can recover your session with `screen -DRR`.
    - You should ***stop*** your AWS instance if you are going back and resume your work in a few hours or days. Your data will be preserved but you will be charged for the EBS storage.
    - Stopped instances still accumulate small fees due to storage. If you are completely done with your instance, you should therefore terminate it. After termination, ***data and any setup on the instance will be completely lost***.
- We recommend you to develop on your own computer and perform experiments on AWS to save your credits. You can upload your code to AWS with `scp`, `rsync`, or `git`.

- You are recommended to use ***private*** repositories provided by GitHub to back up your code. ***Never put your code in a public repo to avoid potential plagiarism.*** To check in your code to a private GitHub repo, create a repo first.

```
git branch -m upstream
git checkout -b main
... // your modifications
git add omp.cpp omp-blocked.cpp
git commit -m "lab1: first version"  # change commit message accordingly
# please replace the URL with your own URL
git remote add origin git@github.com:YourGitHubUserName/your-repo-name.git
git push -u origin main
```

- You are recommended to `git add` and `git commit` often so that you can keep track of the history and revert whenever necessary. If you move to an AWS instance to perform the experiments, just `git clone` your repo.
- If you want to perform both tests, run `make test` to re-compile and test your code. If `make test` fails, it means your code produces the wrong result.
- You can run the sequential GEMM by `make gemm && ./gemm sequential`. Be aware that this is very slow (~10 mins on `m5.2xlarge` instances).
- ***Make sure your code produces correct results!***

Submission

You need to report the performance results of your OpenMP implementation on an `m5.2xlarge` instance. Please express your performance in GFlops and the speed-up compared with the sequential version. In particular, you need to submit a brief report which summarizes:

- The results comparing the sequential version with the two parallel versions on three different problem sizes (1024^3, 2048^3, and 4096^3, change in "`lib/gemm.h`"). If you get significantly different speedup numbers for the different sizes, please explain why.
- `omp-blocked` only: For the problem size 4096^3, please quantify the impact of each optimization, including the block size selection and optimizations you performed.
- `omp-blocked` only: For the problem size 4096^3, please report the scalability of your optimized code with different numbers of threads, including 1, 2, 4, etc, and discuss the result. (hint: please find out how many threads are supported on `m5.2xlarge`).
- A discussion of your results.
- *Optional*: The challenges you faced, and how you overcame them.

You also need to submit your optimized kernel code. Do not modify the testbed code `main.c`, `only omp.cpp and omp-blocked.cpp` source files can contribute to your code's correctness and and performance. Please submit on Gradescope. You will be prompted to enter a Leaderboard name when submitting, please enter the name you want to show to your classmates. You can submit as many times as you want before the deadline to improve your performance. Your final submission should contain and only contain these files individually:

```
├ omp.cpp
├ omp-blocked.cpp
└ lab1-report.pdf
```

Do not separate your code into multiple files. File `lab1-report.pdf` must be in PDF format exported by any software. You could also submit a zip file by copying your `lab1-report.pdf` to the `lab1` directory and `make zip`. Gradescope will automatically extract the files from the zip file and it is expected behavior.

Grading Policy

Your submission will be automatically sent to our AWS Batch queue (m5.2xlarge) for evaluation on Submission Format, Correctness, and Performance. Your last Submission Format score and Performance marks will be final if there is no cheating. The correctness score will only be **partially displayed** before the deadline and may be adjusted during the manual grading. There are other correctness tests of your solution which are not included in the starter kit repo. You can see the performance of other students on the Leaderboard. Although rare, you could request for manual regrade if there is a significant discrepancy between the performance on your instance and Gradescope.

Submission Format (10%)

Points will be deducted if your submission does not comply with the requirement. In the case of missing reports, missing codes, or compilation errors, you will receive 0 for this category.

Correctness (50%)

Please check the correctness of your ***parallel*** and ***parallel-blocked*** implementation with different problem sizes, including but not limited to 1024^3, 2048^3, and 4096^3, each worth 25%. Your code of **parallel-blocked** will be inspected -- if it is not a blocked implementation, you will receive 0.

Performance (25%)

Your performance will be evaluated based on the performance of problem size 4096^3. The performance point will be added only if you have a correct implementation, so please prioritize correctness over performance. Your performance will be evaluated based on the ranges of throughput (GFlops). We will set ranges after evaluating all submissions and assign the points as follows (Ranges A+ and A++ will be defined after all the submissions are made):

- Range A++, better than Range A+ performance: 25 points + 5 points (bonus)
- Range A+, better than Range A performance: 25 points + 3 points (bonus)
- Range A GFlops [92, 121): 25 points
- Range B GFlops [63, 92): 20 points
- Range C GFlops [34, 63): 15 points
- Range D GFlops [5, 34): 10 points
- Speed up lower than range D [0, 5): 5 points
- Slowdown: 0 points

Report (15%)

Points may be deducted if your report misses any of the sections described above.

Academic Integrity

All work is to be done individually, and any sources of help are to be explicitly cited. Any instance of academic dishonesty will be promptly reported to the Office of the Dean of Students. Academic dishonesty, includes, but is not limited to, cheating, fabrication, plagiarism, copying code from other students or from the internet, or facilitating academic misconduct. We'll use automated software to identify similar sections between **different student programming assignments**, against **previous students' code**, or against **Internet sources**. Students are not allowed to post the lab solutions on public websites (including GitHub). Please note that any version of your submission must be your own work and will be compared with sources for plagiarism detection.

Late policy: Late submission will be accepted for **24 hours** with a 10% penalty. No late submission will be accepted after that (you lost all points after the late submission time).