Project 2

Introduction

In this project, we are asked to implement an efficient dense matrix multiplication, which is frequently used in deep learning models. Matrix multiplication plays an increasingly important role in today's AI landscape, serving as a fundamental component of deep neural networks (DNNs), particularly with the development of large language models (LLMs).

Compilation and Execution

To compile the program, please do the following steps:

```
cd project2
mkdir build && cd build
cmake ..
make -j4
```

After compilation, in order to batch process the project in order to get the execution time, you can simply `sbatch` at the project root directory:

```
cd ..
sbatch ./src/sbach.sh
```

The result is stored in `Project2-Results.txt`. You can use `vim` to open it:

```
vim Project2-Results.txt
```

To get the expected output image for each program individually, you can run the compiled file under `build` directory separately:

```
cd build
# Naive
./src/naive /path/to/matrixA /path/to/matrixB /path/to/multiply_result
# Memory Locality
./src/locality /path/to/matrixA /path/to/matrixB /path/to/multiply_result
# SIMD
./src/simd /path/to/matrixA /path/to/matrixB /path/to/multiply_result
# OpenMP
./src/openmp $thread_num /path/to/matrixA /path/to/matrixB
/path/to/multiply_result
# MPI
mpirun -np $process_num ./src/mpi $thread_num_per_process /path/to/matrixA
/path/to/matrixB /path/to/multiply_result
```

In order to get perf result, you can simply run the following bash script:

```
chmod +x perf.sh
./perf.sh
```

which runs the following commands:

```
#!/bin/bash

mkdir -p ./perf_results

# Naive
srun -n 1 --cpus-per-task 1 perf record -e cpu-cycles,cache-misses,page-faults -
g -o ./perf_results/naive_1024.data ./build/src/naive ./matrices/matrix5.txt
./matrices/matrix6.txt ./result.txt
srun -n 1 --cpus-per-task 1 perf record -e cpu-cycles,cache-misses,page-faults -
g -o ./perf_results/naive_2048.data ./build/src/naive ./matrices/matrix7.txt
./matrices/matrix8.txt ./result.txt

# Locality
srun -n 1 --cpus-per-task 1 perf record -e cpu-cycles,cache-misses,page-faults -
g -o ./perf_results/locality_1024.data ./build/src/locality
./matrices/matrix5.txt ./matrices/matrix6.txt ./result.txt
srun -n 1 --cpus-per-task 1 perf record -e cpu-cycles,cache-misses,page-faults -
g -o ./perf_results/locality_2048.data ./build/src/locality
./matrices/matrix7.txt ./matrices/matrix8.txt ./result.txt

# SIMD
srun -n 1 --cpus-per-task 1 perf record -e cpu-cycles,cache-misses,page-faults -
g -o ./perf_results/simd_1024.data ./build/src/simd ./matrices/matrix5.txt
./matrices/matrix6.txt ./result.txt
srun -n 1 --cpus-per-task 1 perf record -e cpu-cycles,cache-misses,page-faults -
g -o ./perf_results/simd_2048.data ./build/src/simd ./matrices/matrix7.txt
./matrices/matrix8.txt ./result.txt

# OpenMP
for threads in 1 2 4 8 16 32; do
    srun -n 1 --cpus-per-task ${threads} perf record -e cpu-cycles,cache-
misses,page-faults -g -o ./perf_results/openmp_1024_${threads}.data
./build/src/openmp $threads ./matrices/matrix5.txt ./matrices/matrix6.txt
./result.txt
    srun -n 1 --cpus-per-task ${threads} perf record -e cpu-cycles,cache-
misses,page-faults -g -o ./perf_results/openmp_2048_${threads}.data
./build/src/openmp $threads ./matrices/matrix7.txt ./matrices/matrix8.txt
./result.txt
done

# MPI
for processes in 1 2 4 8 16 32; do
    threads=$((32 / processes))
    mpirun -np $processes perf stat -e cpu-cycles,cache-misses,page-faults -o
./perf_results/mpi_1024_${processes}.data ./build/src/mpi $threads
./matrices/matrix5.txt ./matrices/matrix6.txt ./result.txt
    mpirun -np $processes perf stat -e cpu-cycles,cache-misses,page-faults -o
./perf_results/mpi_2048_${processes}.data ./build/src/mpi $threads
./matrices/matrix7.txt ./matrices/matrix8.txt ./result.txt
done
```

After that, you can use `perf report` to view the results under the `perf_results` folder. Note that for MPI, since `perf stat` is used, you can use `vim` to see the results.

Parallel Principle

Similar to project 1, for all the methods except SIMD, they all exploit SPMD paradigm and DLP, since they execute the same program but with different data elements.

For SIMD, it focuses on the parallelism on a single operation taking effect in registers of the CPU, since each register is able to contain more than one integer, floating point, etc. It also makes use of DLP to speed up.

Optimization

1. For locality, I use three optimization methods. Firstly, I tile each big matrix into small matrices with a tile size of 64 int length. It can decrease cache miss. Secondly, I do matrix transpose for matrix B and expand the 2D matrix into a 1D array. This can maximize spacial locality, since it only needs to load the matrix consecutively instead of loading different columns and lines. Lastly, I temporally store the result matrix value. Since the innermost loop do multiplication for only one element in the result matrix, we can temporarily store that element inside the register outside the loop, instead of keeping reading and writing the value from the cache.
2. For SIMD, I first use `_mm_malloc` to allocate aligned memory blocks for both input matrices. Then I do parallel multiplication and reduced summation using SIMD registers.
3. For OpenMP, I use static schedule method for multithreading, and also do parallelism for the first two loops in order to distribute tiles.
4. For MPI, I split the rows of the result matrix into different processes, such that each process execute mutual exclusive rows.

Result

The result is shown below:

Methods	**Matrices 1024*1024**	**Matrices 2048*2048**
Memory Locality	581 ms	4646 ms
SIMD + Memory Locality	136 ms	1056 ms
OpenMP + SIMD + Memory Locality (32 threads)	42 ms	173 ms
MPI + OpenMP + SIMD + Memory Locality (total 32 threads)	36 ms	162 ms

The speedup factors are shown as followed, using $S(p) = \frac{t_s}{t_p}$ and Naive performance (8165 ms and 89250 ms) as a baseline.

Methods	**Matrices 1024*1024**	**Matrices 2048*2048**
Memory Locality	14.05	19.21
SIMD + Memory Locality	60.04	84.52
OpenMP + SIMD + Memory Locality (32 threads)	194.40	515.90
MPI + OpenMP + SIMD + Memory Locality (total 32 threads)	226.81	550.93

The efficiencies are shown below. It has a decreasing trend when the thread or process number becomes larger due to the overheads.

Methods	Matrices 1024*1024	Matrices 2048*2048
Memory Locality	140.5%	192.1%
SIMD + Memory Locality	600.4%	845.2%
OpenMP + SIMD + Memory Locality (32 threads)	60.75%	161.22%
MPI + OpenMP + SIMD + Memory Locality (total 32 threads)	70.88%	172.17%

Result Analysis

We can see the locality method surpasses the naive baseline for nearly 7.5 seconds for 1024 * 1024 matrices, and 85 seconds for 2048 * 2048 matrices relatively. With the help of SIMD, the time reduces to 1/5. After using multithreading, the time even decreases to 1/3 for 1024 * 1024 matrices and 1/6 for 2048 * 2048 matrices. While there seems to be decreases in MPI as well, the speed up is not that obvious.

Profiling Results and Analysis

The perf result is shown below:

Matrices 1024*1024

Methods	Cache Misses	Page Faults
Naive	20k	14
Memory Locality	775	7
SIMD + Memory Locality	661	13

OpenMP + SIMD + Memory Locality	Cache Misses	Page Faults
1 thread	668	13
2 threads	745	13
4 threads	746	19
8 threads	877	44
16 threads	970	44
32 threads	1k	52

MPI + OpenMP + SIMD + Memory Locality (total 32 threads)	Cache Misses	Page Faults
1 process	842718	7054
2 processes	666343	7170
4 processes	496100	7078
8 processes	600427	8546
16 processes	580307	9697
32 processes	656163	10133

Matrices 2048*2048

Methods	Cache Misses	Page Faults
Naive	359k	1k
Memory Locality	5k	1k
SIMD + Memory Locality	4k	239

OpenMP + SIMD + Memory Locality	Cache Misses	Page Faults
1 thread	4k	234
2 threads	5k	256
4 threads	5k	327
8 threads	7k	335
16 threads	9k	391
32 threads	9k	348

MPI + OpenMP + SIMD + Memory Locality (total 32 threads)	Cache Misses	Page Faults
1 process	7,736,302	22,457
2 processes	3,020,291	22,557
4 processes	3,315,905	24,261
8 processes	3,630,117	27,199
16 processes	2,041,654	26,729
32 processes	2,536,197	38,538

1. For locality, such a great improvement comes from a large decrease in cache misses from 20k to 775 for 1024 * 1024 matrices, and from 359k to 5k for 2048 * 2048 matrices due to the tiling process. Meanwhile, page faults seem to have little change. For SIMD, the decrease in cache misses is even large, but the page fault decrease is obvious in 2048 * 2048 matrices

from 1k to 239. This might be because of the optimization in aligned method of input matrices allocation.

2. There is an slight increase of both cache misses and page faults in OpenMP while the thread number increases. As more threads are added, multiple threads may try to access the same memory locations or cache lines. This increases the likelihood of cache contention, where threads evict each other's data from the cache. For MPI, we can see a huge increase in both cache misses and page faults. But there is no clear relation between number of processes and these two, since the threads number is fixed. Such large increase might be due to message passing through different cores between processes.